



SPERRY
5000 Series

Device Driver Guide
For 5000/20
through 5000/50



SPERRY 5000 Series

Device Driver Guide

5000 Series System Library
UP-12230

This document contains the latest information available at the time of preparation. Therefore, it may contain descriptions of functions not implemented at manual distribution time. To ensure that you have the latest information regarding levels of implementation and functional availability, please consult the appropriate release documentation or contact your local Sperry representative.

Sperry reserves the right to modify or revise the content of this document. No contractual obligation by Sperry regarding level, scope, or timing of functional implementation is either expressed or implied in this document. It is further understood that in consideration of the receipt or purchase of this document, the recipient or purchaser agrees not to reproduce or copy it by any means whatsoever, nor to permit such action by others, for any purpose without prior written permission from Sperry.

SPERRY is a registered trademark of
the Sperry Corporation.

UNIX is a trademark of AT&T Bell Laboratories.

MULTIBUS is a trademark of Intel Corp.

MC68000 and MC68010 are trademarks of Motorola Corp.

NEC is a trademark of NEC Information Systems

©1986 - SPERRY CORPORATION
PRINTED IN U.S.A.

Portions of this material are copyrighted ©
by AT&T Technologies, Inc.,
and by NCR Corporation, Dayton, Ohio,
and are printed with their permission.

This documentation is based in part on the Fourth Berkeley Software Distribution under license from The Regents of the University of California. We acknowledge the following individuals and institutions for their role in its development: William Joy; J.E. Kulp of I.I.A.S.A., Laxenburg, Austria; Ken Arnold; Jim Kleckner; Ed Pelegri-Llopart; Howard Katseff; Charles Haley; Eric Shienbrood; John Foderaro; Geoffrey Peck; Mark Horton.

Contents

Chapter 1 Overview	1-1
Introduction.....	1-1
Content	1-1
Chapter 2 Device Driver Overview	2-1
Introduction.....	2-1
Program/Driver Interface Overview	2-2
Device Driver Block Description	2-3
Chapter 3 Creating a New Kernel	3-1
Overview.....	3-1
Installation	3-1
File/Procedures Description	3-4
Purpose of the Configuration Program	3-4
Master File	3-5
Dfile(Configuration File)	3-12
Miscellaneous Files	3-15
Makefile Procedures	3-16
Archiving the Driver.....	3-17
Making an Inode	3-18
System File Entries	3-18
Chapter 4 Process Management	4-1
Overview.....	4-1
Process Definition	4-1
Process States	4-3
Sleep and Wakeup	4-5
Chapter 5 A Pseudo Device Driver	5-1
Overview.....	5-1
I/O System Control Flow.....	5-3
Device Identification	5-3
Device Creation - /etc/mknod.....	5-4
Character Device Switch Table.....	5-5
System Include Files	5-8
Sd - Declarations	5-8
Device Driver Open and Close Routine.....	5-10
Sd Definition - sdopen	5-11
Related Include Files	5-12

Contents

Sdopen Listing	5-15
Sd Definition - sdclose	5-15
Sdclose Listing	5-16
Accessing User Memory Space (ioctl functions)	5-16
Sd Definition - sdioctl	5-17
Sdioctl Listing	5-20
Process Management, U Structure	5-20
Data Movement and the U Structure (Passc Cpass)	5-22
Sd Definition - sdread	5-23
Sdread Listing	5-24
Sd Definition - sdwrite	5-25
Sdwrite Listing	5-25
Creating the New Kernel	5-26
Sd Listing	5-28
 Chapter 6 Hardware Device Control	6-1
Device Control	6-1
Hardware Controller Registers	6-1
Reading and Writing Device Registers	6-1
Byte Ordering	6-3
Multibus Address Space Available to Customers	6-7
Bus Vectored Interrupts	6-8
Interrupt Vectors Available to Customers	6-9
 Chapter 7 High Performance Serial I/O Board	7-1
Overview	7-1
HPSIO Features	7-3
HPSIO Interface	7-3
Mailbox Register	7-6
Interrupt Vector Register	7-7
Interrupt Status/Arbitration Register	7-8
I/O Parameter Block	7-9
IOPB Interface	7-10
I/O Completion - Incoming IOPB's	7-11
HPSIO Communication Concepts	7-12
Channel Initialization	7-12
Output	7-15
Input	7-16
Character Acknowledgement	7-17

Chapter 8 A Character Device Driver	8-1
Overview	8-1
Driver Overview	8-1
Control Flow Example	8-3
Data Flow Example	8-3
Lower Half Output Logic	8-3
Chopen Definition	8-4
Initial Requirements Section	8-4
Protect Section	8-4
Channel Initialization Section	8-5
Unprotect Section	8-5
Chopen Listing	8-5
Chread Definition	8-7
Chread Listing	8-9
Chwrite Definition	8-9
Chwrite Listing	8-9
Chclose Definition	8-10
Chinit Definition	8-11
Chinit.c Listing	8-11
Chstart Definition	8-12
Chstart Listing	8-12
Chintr Definition	8-13
Channel Initialization (CFCHANINIT)	8-14
Character Typed (CFCHARACK)	8-14
Output Acknowledgement (CFOUTPUT)	8-14
Chintr Listing	8-15
Master/Dfile Entries	8-17
Creating/Testing the Kernel	8-18
Complete Source Listing	8-20
 Chapter 9 A TTY Driver	 9-1
Introduction	9-1
Example of LDR Functions	9-2
General Control Flow	9-4
NEC Terminal Controller	9-5
Data Flow Diagram	9-8
Control Flow	9-9
Line Switch Table	9-10
TTY Structure	9-10
RS-232-C Hardware Signals: DTR and DCD ...	9-14
Necopen Definition	9-14

Necopen Listing	9-14
Device Driver Interrupt Service Routines.....	9-15
T_rbuf.....	9-16
T_tbuf	9-17
Receiver/Transmitter Listing	9-23
Input/Output Flow Control	9-24
Receive ISR Listing	9-25
XON/XOFF Protocol for Input Flow Control	9-28
Transmitter ISR Listing	9-29
Ready/Busy Flow Control	9-30
Parity.....	9-30
Receiver ISR Listing	9-31
ISTRIP.....	9-32
Receiver ISR - Final Version	9-33
LDR to Driver Communication	9-34
Necproc Listing.....	9-36
Driverproc(tp, T_OUTPUT)	9-37
Pnec.output.c Listing.....	9-38
Driverproc(tp, T_BLOCK) Listing	9-39
Necproc(tp, T_BLOCK) Listing	9-39
Driverproc(tp, T_UNBLOCK)	9-40
Pnec.unblock.cf Listing.....	9-40
Driverproc(tp, T_SUSPEND-tp, t_RESUME) ...	9-41
Driverproc(tp, T_BREAK)	9-41
Pnec.break.c Listing	9-42
Proc Command - Flag Affected	9-43
Close Routine.....	9-44
Pnec.close.c Listing	9-44
IOCTL Routine	9-45
Error Handling	9-45
 Chapter 10 A Virtual Disk Driver	 10-1
Introduction to Disk Drivers.....	10-1
Block Handling Routines	10-1
Buffers and Buffer Headers	10-3
Driver I/O Queue	10-6
Block Device Interface	10-7
Bdevsw	10-8
Virtual Disk Driver Description	10-8
Vdopen Definition	10-9
Vdopen Listing	10-9

Virtual Driver Strategy Definition	10-9
Vdstrategy Listing	10-10
Virtual Disk Driver Complete Listing	10-11
Character Device Portion of the Driver	10-15
Character Read and Write Listing	10-15
Physio I/O	10-15
Physical I/O Strategy Definition	10-16
PhysicalI/O Strategy Listing	10-16
Memory Management.....	10-18
Virtual Disk Driver Init Definition	10-24
Init Listing	10-24
Strategy Definition	10-25
Physical Device Implementation	10-29
Read.....	10-29
Complete Block and Physical Device Driver ..	10-31
 Chapter 11 Miscellaneous Procedures.....	 11-1
Power Failure Recover	11-1
Bus Errors and User Memory Space.....	11-3
Error Logging	11-7
Communicating Errors to Application Processes	11-7
Logging Hardware Events	11-8
Error Include Files.....	11-9
Character Device Error Logging	11-9
Block Device Driver Error Logging.....	11-12
Diagnostic System Call	11-16
 Appendix A HPSIO Edited Functional Specification	A-1
Controller Port Definitions	A-1
CIOPB Structure	A-4
 Appendix B Clists	 B-1
Device Driver Clists	B-1
Clist Definitions	B-2
Cblocks and Clists.....	B-2
Cblock Free List.....	B-3
Character Buffered I/O Diagrams	B-4
getc	B-5
putc	B-10
getc f	B-11
putc f	B-13

Contents

getc	B-15
putc	B-17
getc	B-18
putc	B-20
Appendix C Bootable Media	C-1
Overview	C-1
Hard Disk Boot	C-1
Flex Disk Boot	C-2
Streaming Tape	C-2
Appendix D Kernel Calls	D-1
Overview	D-1
Appendix E System Calls	E-1
Overview	E-1

Chapter 1.

Overview

INTRODUCTION

This guide provides assistance for integrating a device driver into your SPERRY 5000 series operating system. Information in this guide applies to all currently supported releases of the SPERRY 5000 Operating System for the 5000/20, 5000/40, and 5000/50.

This guide is written for programmers, analysts, and other support personnel. Before using this guide, you should be familiar with your operating system, be knowledgeable in the "C" programming language, and be familiar with the hardware device and its interface.

CONTENTS

The information in this guide

- Identifies the entries required for the system files.
- Identifies the primary driver structures.
- Identifies the steps for integrating the driver into the system.
- Provides example driver programs.
- Explains driver related routines such as open, strategy, close, and interrupt service.
- Describes the driver-related system calls and driver-related kernel calls.

Chapter 2.

Device Driver Overview

INTRODUCTION

A device driver is the part of the kernel that controls hardware devices. Control includes such functions as sending control information, interpreting statuses, processing interrupts, writing data, and reading data. After a device driver is integrated into the kernel, it becomes a high level interface between an application program and hardware device.

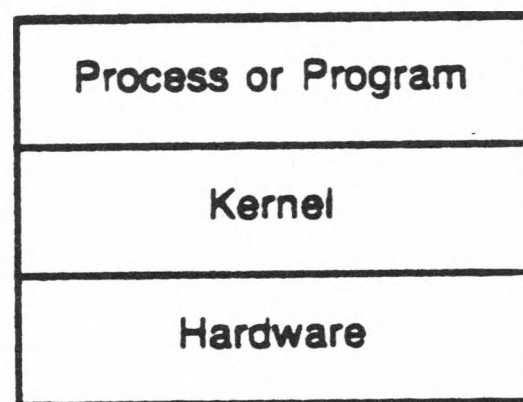


Figure 1. Layer Overview

As shown in Figure 1, the kernel is the layer of software between a process or program and the actual hardware. The kernel permits the referencing of devices by names and permits device control using high level system calls. The kernel provides many other functions and structures for a consistent and flexible interface between the application and the device driver which is part of the kernel.

PROGRAM/DRIVER INTERFACE OVERVIEW

High level system calls provide the method used by a program to interact with a driver. The I/O system calls available to a program are *open*, *close*, *read*, *write*, *ioctl*, and *lseek*. These system calls cause a trap to the kernel. The kernel executes code on behalf of the calling process. Data structures, managed by the kernel, are updated and a device driver function is called.

As shown in Figure 2, the kernel divides I/O into two layers:

- Device independent processing -- the same code is executed for all devices.
- Device dependent processing -- calls are made from the device independent code to a device driver.

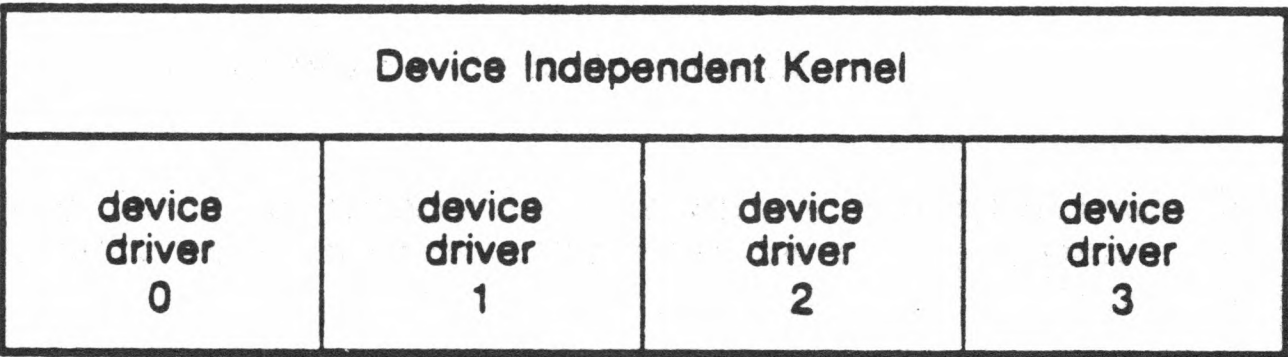


Figure 2. Device Driver Interface

The device independent interface between the kernel and the driver consists of a device switch table. This table specifies the interface routines available for the devices.

Each device driver is referenced by its position in the device switch table. The first entry is referred to as device driver 0, the second is device driver 1, etc. A

special file, representing a device, contains a major device number that identifies the device driver. The entry in the device switch table specifies the routines available for the selected device. This method permits the operating system to resolve external references (binding) between the device driver routines and user programs at run time. This provides for a structured interface while maintaining system flexibility.

For example, an open system call might look like this:

```
open(1); /* open device number 1, the terminal */
```

If the relative location of the device within the device switch table changes, all programs using the device must be recompiled. To avoid this, another level of indirection is needed:

```
open("/dev/tty"); /* open the terminal */
```

The name `/dev/tty` is used to determine the major device number which provides the indexing into the device switch table. An entry in the table accesses the open routine for the device.

NOTE: The same name could correspond to major device number three on one system and major device number five on another system.

DEVICE DRIVER BLOCK DESCRIPTION

Device drivers are partitioned into two segments: upper half and lower half. As shown in Figure 3, user processes access the upper half through system calls that reference the device switch tables. The upper half routines don't manipulate devices directly.

half routines don't manipulate devices directly. Instead, they enqueue requests for data transfer and rely on routines in the lower half to perform the transfer.

Except for starting the lower half routines, the only communication between the upper and lower halves of the device driver is performed using queues. The lower half routines are typically interrupt driven.

In summary, the buffering and the interrupt processing permits data to be transferred between the application and the buffers at processor speeds, and between the device and the buffers at slow device speeds.

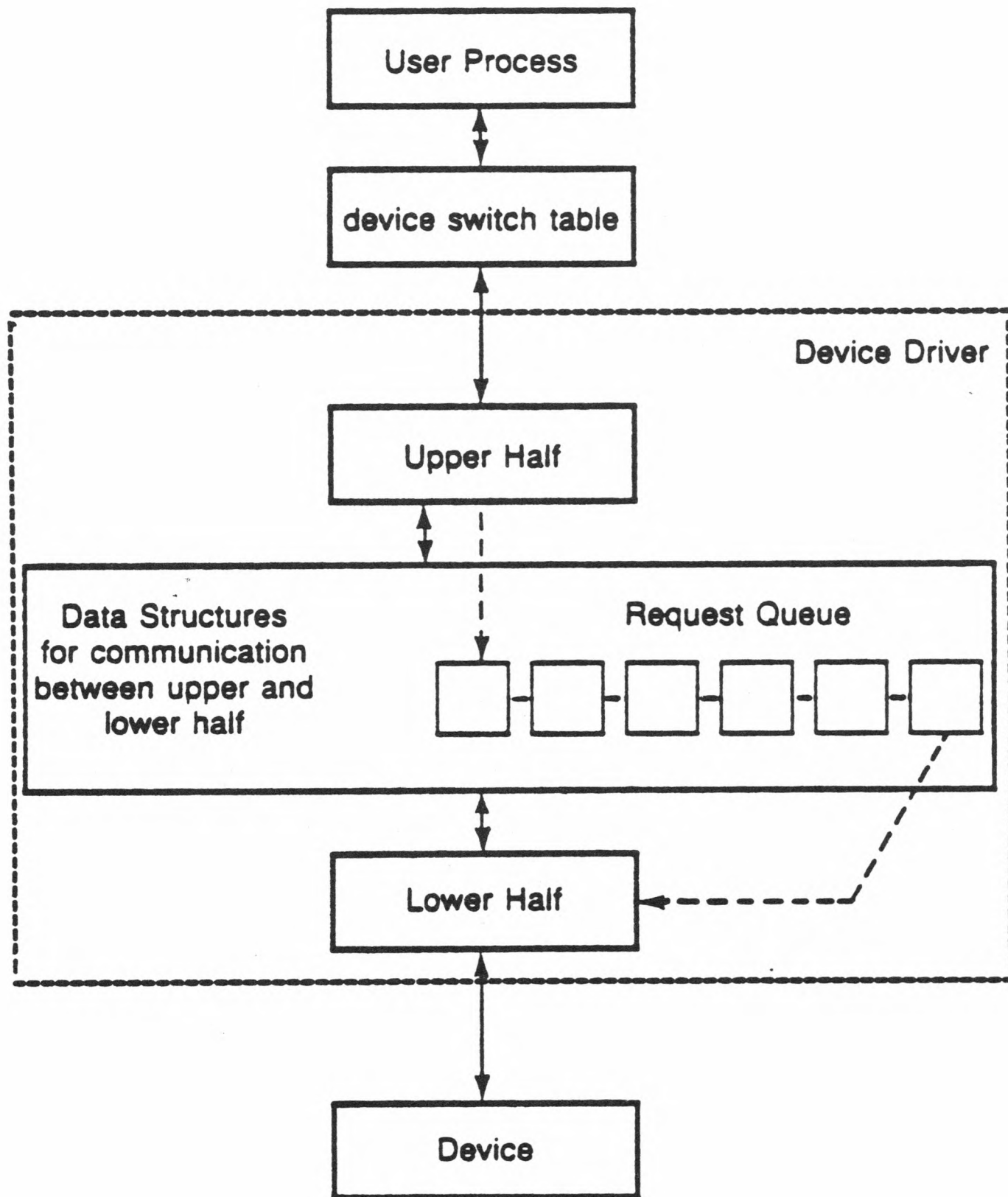


Figure 3. Driver Overview

Chapter 3.

Creating a New Kernel

OVERVIEW

This chapter provides the information relating to the system configuration files and the procedures for creating a new kernel. The information introduces the files required for a device driver and introduces other driver concepts as they relate to entries in the files.

We recommend you have available a printed copy of the files identified in the following list to identify the driver related entries for the current system configuration. Make any changes to the */etc/master* and */kernel/sperry/cf/5.2.cf*, and *name.c* files in your home directory. Consequently, the working kernel configuration isn't changed.

- */etc/master*
- */kernel/sperry/cf/5.2.cf*
- */kernel/sperry/cf/conf.c*
- */kernel/sperry/cf/univec.c*

INSTALLATION

Here are the steps required to include a device driver into the kernel. Figure 1 identifies the primary elements involved in the installation procedures.

- Write and compile the device driver (*newdevice.c*).
- Modify the *dfile* (configuration file).
- Execute a make file that:

+ Executes the *config* program.

- + Compiles the output of the *config* program.
 - + Links the kernel libraries with the new driver and the compiled output of the *config* program.
-
- Make an *inode* (special file) for the device.
 - Include in the */sys/cf* and */etc/master* files the same entries made in *5.2.cf* and *master*.
 - Archive the *newdevice.o* code in the */sperry/kernel/lib2* file.

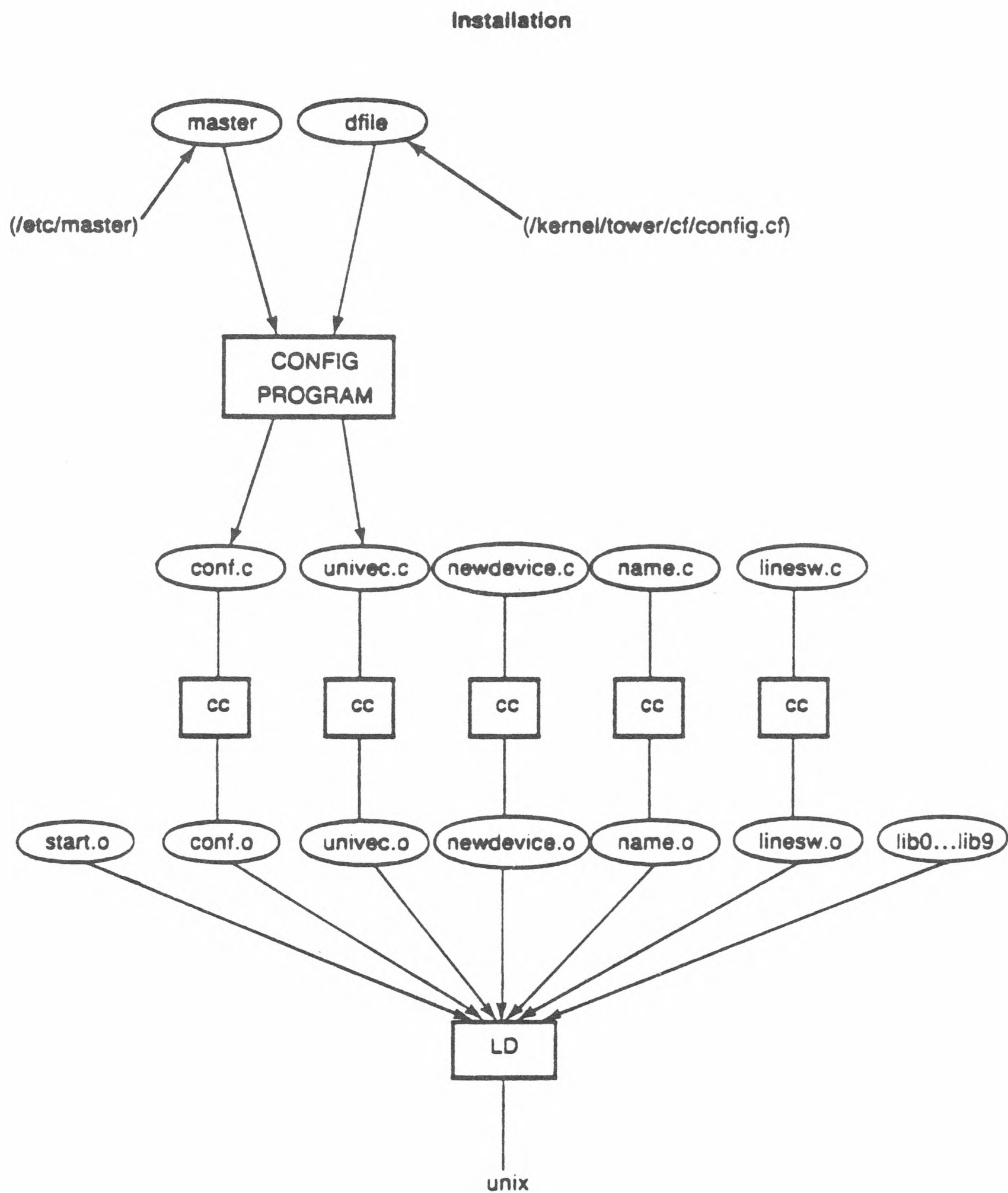


Figure 1. Creating a Kernel

FILE/PROCEDURES DESCRIPTION

This section describes the required files and makefile procedures.

Purpose of the Configuration Program

The purpose of the *config* program is to create two C programs according to a set of specifications entered in *dfile* (5.2.cf) file. The *config* program has two input files, *master* and *dfile*, and two output files, *conf.c* and *univec.c*.

The *master* file describes all of the device drivers permitted on the system. *Config* uses information from the *master* file to carry out directives in the configuration file *dfile*. *Config* creates two output files: *conf.c* contains all tunable data structures including the block and character device switch tables; *univec.c* contains the interrupt vector table. This table provides the interrupt vector addresses for the hardware related interrupt service routines.

Using Figure 1 as an example, the format of the *config* command is:

```
/kernel/sperry/cf/config -m master 5.2.cf
```

The *config(1M)* command is described in the *Administrator Reference*, UP-11761.

Master File

The *master* file (an *augmented* version of */etc/master*) contains information used by the *config* program to generate configuration files [see *config(1M)*]. The *master* file consists of three parts:

- Part 1 contains a line entry for each device driver.
- Part 2 optionally contains device name aliases.
- Part 3 contains tunable system parameters. These parameters don't directly relate to device drivers but can control resources used by device drivers.

A line that contains a dollar sign (\$) in column 1 separates each part of the file.

Master File Part 1

A full description of the *master* file is found in *master(4)* of the *Programmer Reference*, UP-11762. Part 1 of the *master* file has one line per device driver. Each line has 10 required fields and 3 optional fields. The following two lines are example entries for a disk and a console driver.

1	2	3	4	5	6	7	8	9	10	11	12	13
dkx	4	1477	214	dk	4	25	27	4	6	stat1	stat2	stat3
crt	1	677	4	cr	1	0	28	8	6	tty		

Figure 2. Master file entries

This list describes the fields shown in Figure 2.

Field 1

Device name (8 chars. maximum); must be unique. This field is a key used in the first part of the *dfile* to identify this line in the *master* file.

Field 2

Interrupt vector count and naming conventions for the interrupt service routines. These are used to build the *univec.c* file. Refer to the Field 2 text for entry descriptions.

Field 3

Device mask. Refer to the Field 3 text for entry descriptions.

Field 4

Device mask 2. Refer to the Field 4 text for entry descriptions.

Field 5

Driver (handler) prefix; defines <hp> and is used in the other field descriptions.

Field 6

Device register address space size in decimal. This field is the amount of *Multibus* address space needed by the device. The *dkx* driver has a value of 4 showing that 4 bytes of on-board registers are accessible by the disk driver. The *crt* driver has a value of 1 showing a one byte addressable register. This field must be non zero in order to generate a <hp>_addr line in *conf.ck*. This field is used in the configuration test that avoids *Multibus* I/O address collisions.

Field 7

Major device number for the block device.

Field 8

Major device number for the character device.

Field 9

Maximum and default number of subdevices for the controller.

Field 10

Maximum bus request level. This field is used to limit the variable entered in field 4 of the *dfile* part 1.

Optional fields

The last three fields are optional. Each field can contain the name of a data structure needed by the device driver. The configuration program makes this declaration in the file *config.c*: *struct value <hp>-value[N]*. Value is the string of characters in field 11, 12 or 13. <hp> is the handler prefix. N is an integer value that is the same as the total number of subdevices (for example, number of controllers times subdevices per controller). In the *dkx* example, this declaration is created: *struct stat1 dk_stat1[4];*.

NOTE: Make the declaration *struct stat1* available to the *conf.c* file. Make the entry in */usr/include/sys/space.h*.

Field 2 Description

Here is a description of the interrupt vector count and naming conventions used to generate the file *univec.c*.

- 0 - No interrupts. Used for pseudo devices.
- 1 - Used for non-bus vectored devices to yield interrupt service routine (ISR) names in the format: `<hp><board number>int`; `<hp>` represents the handler prefix (device name).
- 4 - One interrupt with ISR name `<hp>intr`.
- 5 - One interrupt with ISR name `<hp>0int`.
- 10 - Two interrupts with ISR names `<hp>0int` and `<hp>1int`.
- 12 - Three interrupts with ISR names `<hp>0int` through `<hp>2int`.
- 16 - Four interrupts with ISR names `<hp>0int` through `<hp>3int`.
- 20 - Five interrupts with ISR names `<hp>0int` through `<hp>4int`.
- 24 - Six interrupts with ISR names `<hp>0int` through `<hp>5int`.
- 28 - Seven interrupts with ISR names `<hp>0int` through `<hp>6int`.
- 32 - Eight interrupts with ISR names `<hp>0int` through `<hp>7int`.

In the example *dkx*, one bus vectored interrupt causes the function *dkxintr* to be invoked. *Crt* has a non-bus vectored interrupt that causes the function *cr0int* for board 0, *cr1int* for board 1, etc.

Field 3 Description

This field is a device mask. The bit flags turned on by this octal number describe some of the characteristics of a character device driver.

- 01000 - Memory management resources are needed for Direct Memory Access (DMA). Entries in the segment map are reserved by setting this bit.
- 00400 - In-service diagnostic routines exist. A function with the name `<hp>diag` must exist in the driver.
- 00200 - A `tty` structure is included in the `cdevsw` table. The line discipline routines are accessed through the line switch table on system read and write calls instead of through the driver's read and write routines.
- 00100 - An initialization function exists for the device with the name `<hp>init`. This routine is called when the system is booted.
- 00040 - A power fail recovery function exists for the device with the name `<hp>clr`, called after power fail.
- 00020 - An open function exists for the device with the name `<hp>open`. When a user process executes an open system call, this function is called. This function has an entry in the `cdevsw` table.
- 00010 - A close function exists for the device with the name `<hp>close`. When the last close for a minor device is executed by a user process, this function is called. This function has an entry in the `cdevsw` table.
- 00004 - A read function exists for the device with the name `<hp>read`. When a user process executes a read system call, this function is called. This function has an entry in the `cdevsw` table.
- 00002 - A write function exists for the device with the name `<hp>write`. When a user process executes a write system call, this function is called. This function has an entry in the `cdevsw` table.

- 00001 - An ioctl function exists for the device with the name <hp>ioctl. When a user process executes an ioctl system call, this function is called. This function has an entry in the cdevsw table.

In Figure 2, the dkx driver has these characteristics:

01000	Memory management resources needed for DMA.
00400	A dkdiag routine for inservice diagnostics exists.
00040	A dkclr routine for power fail recovery exists.
00020	A dkopen routine exists.
00010	A dkclose routine exists.
00004	A dkread routine exists.
00002	A dkwrite routine exists.
00001	A dkioctl routine exists.

01477	All of the above

The crt driver has these characteristics:

00400	A crdig routine for inservice diagnostics exists.
00200	A tty structure is to be included in cdevsw.
00040	A crclr routine for power fail recovery exists.
00020	A copen routine exists.
00010	A crclose routine exists.
00004	A crread routine exists.
00002	A crwrite routine exists.
00001	A criioctl routine exists.

00667	All of the above

Field 4 Description

This is the device mask 2 field. The bit flags turned on by this octal number describe some of the characteristics of the driver.

- 00400 - The controller is non-bus vectored (for example, autovectored).
- 00200 - Only one device is permitted.
- 00100 - Suppress the count field in the *conf.c* file.
- 00040 - Suppress the interrupt vectors if generated.
- 00020 - Device is required.
- 00010 - Block device.
- 00004 - Character device.
- 00002 - Reserved.
- 00001 - Reserved.

In Figure 2, the dkx driver has these characteristics:

00200	Only one device allowed.
00010	Block device.
00004	Character device.

00214	All of the above

The crt driver has this characteristic:

00004	A character device.
-------	---------------------

Master File Part 2

Part 2 contains lines with 2 fields in each line. Each line defines an alias that can be used in the *dfile*. Part 2 isn't usually used.

- Alias name of device (8 chars. maximum).
- Reference name found as the first field in part 1.

Master File Part 3

Part 3 entries describe the tunable system parameters that aren't used by the driver but can control resources used by the driver.

Dfile (Configuration File)

The *dfile* describes the configuration of the kernel. The first part contains a list of driver related entries to be included in the kernel. The second part shows the devices used for root, pipe, and dump.

```
root devname minor  
pipe devname minor  
dump devname minor
```

Devname is the device name or alias as defined in the *master* file. *Minor* is the subdevice number (in octal). The specification of the swap device is also defined in the line:

```
swap devname minor swaplo nswap
```

Swaplo is a decimal value identifying the first block in the swap area, (value must be 1) and *nswap* is a decimal value specifying the number of blocks in the swap area.

Finally, there are a variable number of lines that can

be used to redefine the tunable parameters set up in the *master* file. These lines have this format:

keyword value

Keyword is the name found in part 3 of the *master* file and *value* is the numeric constant to be used.

Dfile Driver Entries

The lines in Part 1 have this format:

devname intvect baseaddr intlevel subdevs

- *Devname* - Device name or alias defined in field 1 of the *master* file.
- *Intvect* - Interrupt vector offset in octal. The *intvect* is NOT the vector number but is the address of the vector. If more than one vector is generated as configured in field 2 of the *master* file, this is the base interrupt vector.
- *Baseaddr* - Base *Multibus* I/O address of the controller is octal. Must be an even number. Should be in the range 0-177777.
- *Intlevel* - *Multibus* interrupt request level for a bus vectored device. *Intlevel* is the poll address for non-bus vectored devices.
- *Subdevs* - Actual number of subdevices on the controller.

The base address and intlevel are parameters to the driver and don't affect the system.

For example,

dkx 460 172220 4 4

The dkx disk driver interrupt vector location is 460. The *Multibus* base address is 172220. The interrupt

Chapter 3

level is 4. The number of subdevices is 4. For example,

```
*board 0
crt 470 160000 1 8
*board 1
crt 480 161000 2 4
```

The interrupt service routine *crt0int* is invoked when board 0 generates an interrupt. The interrupt service routine *crt1int* is invoked when board 1 generates an interrupt.

There are 12 variables (number of minor devices) defined for each device unless that feature has been disabled by the *master* file.

- Multibus base addresses 16000 and 161000.
- Positions 1 and 2 are used by the boards on the autovector poll lists.
- The first board has 8 sub-devices and the second board has 4 sub-devices.

The following two lines are contained in the *conf.c* file:

```
int dk_cnt = 4;
int cr_cnt = 12;
```

The total number of subdevices is generated for the crt driver.

The following lines would also be generated.

```
int dk_addr[]=  
    {172220, 4, 460};  
int cr_addr[]=  
    {160000, 1, 470,  
    161000, 2, 480};
```

NOTE: These are arrays that contain a line for each controller.

Miscellaneous Files

Miscellaneous files required for creating a kernel are:

newdevice.c

The *newdevice.c* file contains the source code routines for the new device driver. This file typically resides in the user's home directory.

start.o

For the 5000/20 and 5000/40, the *start.o* file relocates the kernel code to a logical starting address of 108000H following a bootload. For the 5000/50 and 32-bit 5000/40, the *start.o* file relocates the kernel code to a logical starting address of E00000H following a bootload. This file resides in */kernel/sperry/ml*. *This must be the first file in the ld(1) command, used to link the kernel.*

linesw.o

The *linesw.o* file is a switch table with line entries that define the line discipline routines (LDRs) that are available for character devices. Each line is indexed by an LDR number. LDR 0 is the only line in the table and identifies the line discipline routines for tty type terminals. This file resides in */kernel/sperry/linesw.c*.

name.c

The *name.c* file contains the user defined header information output to the console when the kernel is booted and includes system name, node name, release, version, and machine name. This file resides in */kernel/sperry/cf/name.c*.

lib0-lib9

These files are the kernel libraries that reside in the */kernel/sperry* directory.

Makefile Procedures

The procedures for creating a new kernel are included in the following make file. The *make(1)* command is used to execute the file.

In this example, the *makefile* is in the user's home directory and the format for the make command is:

make -f Makefile unix

```
/* The Makefile */
```

```
test    =  newdevice.o
```

```
#  define for name.o so boot up console messages are printed out correctly
```

```
SYS =    UTS V
```

```
VER =    yours
```

```
NODE    =    S5000
```

```
REL =    1R1
```

```
#  MACH = 68010 for 5000/20 and 5000/40
```

```
#  MACH = 68020 for 5000/50 and 32-bit 5000/40
```

```
MACH    =    68010
```

```
DEFS    =    -DSYS=\"$(SYS)\" \
```

```
          -DVER=\"$(VER)\" \
```

```
          -DNODE=\"$(NODE)\" \
```

```
          -DREL=\"$(REL)\" \
```

```
          -DMACH=\"$(MACH)\"
```

```
AS      =    as
```

```
CC      =    cc
```

```
#  RELOC = 108000 for 5000/20 and 5000/40
```

```
#  RELOC = E00000 for 5000/50 and 32-bit 5000/40
```

```
RELOC    =    108000
```

```
CFLAGS  =    -K -I. $(INCRT)
```

```
LFLAGS = -R $(RELOC) -N -e susentry
```

```
LIBS = /kernel/sperry
```

```
INCRT = /usr/include
```

```
unix:  conf.o name.o univec.o $(LIBS)/cf/linesw.o $(LIBS)/lib[0-9] $(test)
      ld $(LFLAGS) /kernel/sperry/ml/start.o\
          univec.o\
          conf.o\
          name.o\
          $(LIBS)/cf/linesw.o\
          \
          $(test)\
          \
          $(LIBS)/lib[0-9]\
          $(LIBS)/lib3\
          -o unix\
          2> undefs
```

```
name.o: $(INCRT)/sys/utsname.h
      $(CC) $(CFLAGS) $(DEFS) -c name.c
```

```
conf.o: conf.c newdevice.h
```

```
univec.o: univec.c
```

```
univec.c: 5.2.cf master
      config -m master 5.2.cf
```

```
conf.c: 5.2.cf master
      config -m master 5.2.cf
```

```
newdevice.o:  newdevice.c newdevice.h
```

Archiving The Driver

After compiling and testing the new kernel, the driver must be archived in the kernel. The driver typically resides in */kernel/sperry/lib2*. The command for archiving is:

```
cd /kernel/sperry
```

```
ar rv lib2 newdevice.o
```


The *ar(1)* command is described in the *Administrator Reference, UP-11761*.

Making an Inode

The *mknod* command adds the driver *inode* to the system.

```
/etc/mknod /dev/name [c or b] major minor
```

Name is the name of the device, *c* or *b* specifies the type of device (character or block), *major* is the major number, and *minor* is the minor number. Major and minor entries are the same as the entries in the *master* file.

System File Entries

The */etc/master* and */sys/cf* system files must be updated to include the same entries that are made in *5.2.cf* and *master*.

Chapter 4.

Process Management

OVERVIEW

A device driver performs I/O based on the request of a process. Most I/O in the operating system is interrupt driven. After a process requests I/O, the driver makes a request to suspend the execution of the calling process until the I/O is completed. After the suspension of a process, another process can start executing. Process management controls the starting and suspending of processes.

PROCESS DEFINITION

As shown in Figure 1, a process consists of a program (user code) and the kernel. When the program performs a system call, it transfers control to the kernel.

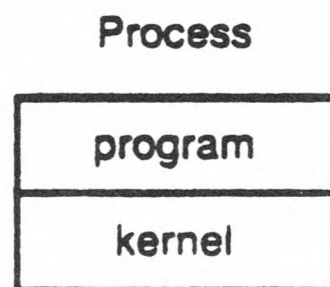


Figure 1. Process components

An important function of the kernel is to manipulate processes. The device driver portion of the kernel plays a major role in controlling processes.

In a multiprocessing system, these conditions describe the relationship between a process and the processor (CPU):

- Many processes typically compete for the processor.
- Only one process may actually be using the processor at one time.
- All processes assume exclusive use of the processor.

At any time, the highest priority process eligible for CPU service is executing. Among processes with equal priority, scheduling is round-robin, where processes are selected one after another so that all members of the set have an opportunity to execute before any member has a second opportunity. The currently active process makes requests for services, such as reading data from a file or writing to a terminal. These requests cause kernel code to be executed.

Figure 2 shows how a processor might be shared between three processes. Each dash represents a time unit.

A process may lose the processor for several different reasons, such as expiration of its time slice, request for a service that can't be immediately satisfied, a more important process being activated, etc. Therefore, each process does not receive an equal time slice.

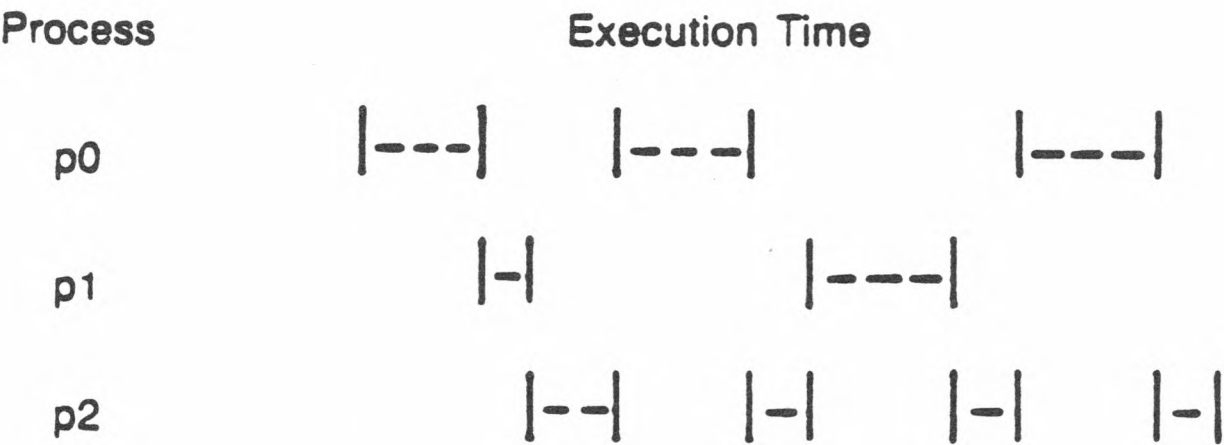
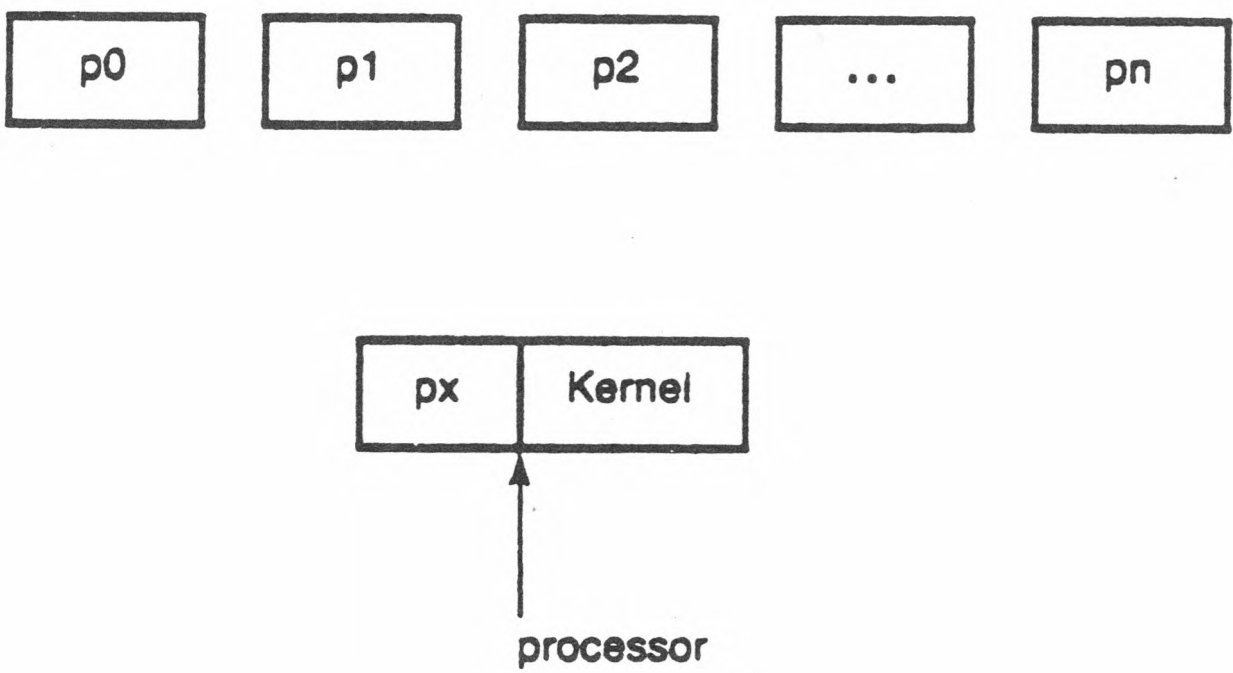


Figure 2. Process timesharing

PROCESS STATES

During its existence, a process goes through a series of discrete states -- ACTIVE, READY, and SLEEP:

- The currently executing process is in the ACTIVE state. Only one process can be in the ACTIVE state at any one time.
- Any process ready to use the processor is in the READY state. Many processes can be in the READY state at the same time. Processes wait on a READY list to compete for the processor.

- Any process waiting for some condition to be satisfied is in the SLEEP state. For example, a process may be asleep waiting for a character to be entered on a terminal. Many processes can be in the SLEEP state.

When a process changes states, it makes a state transition. Figure 3 describes the state transitions.

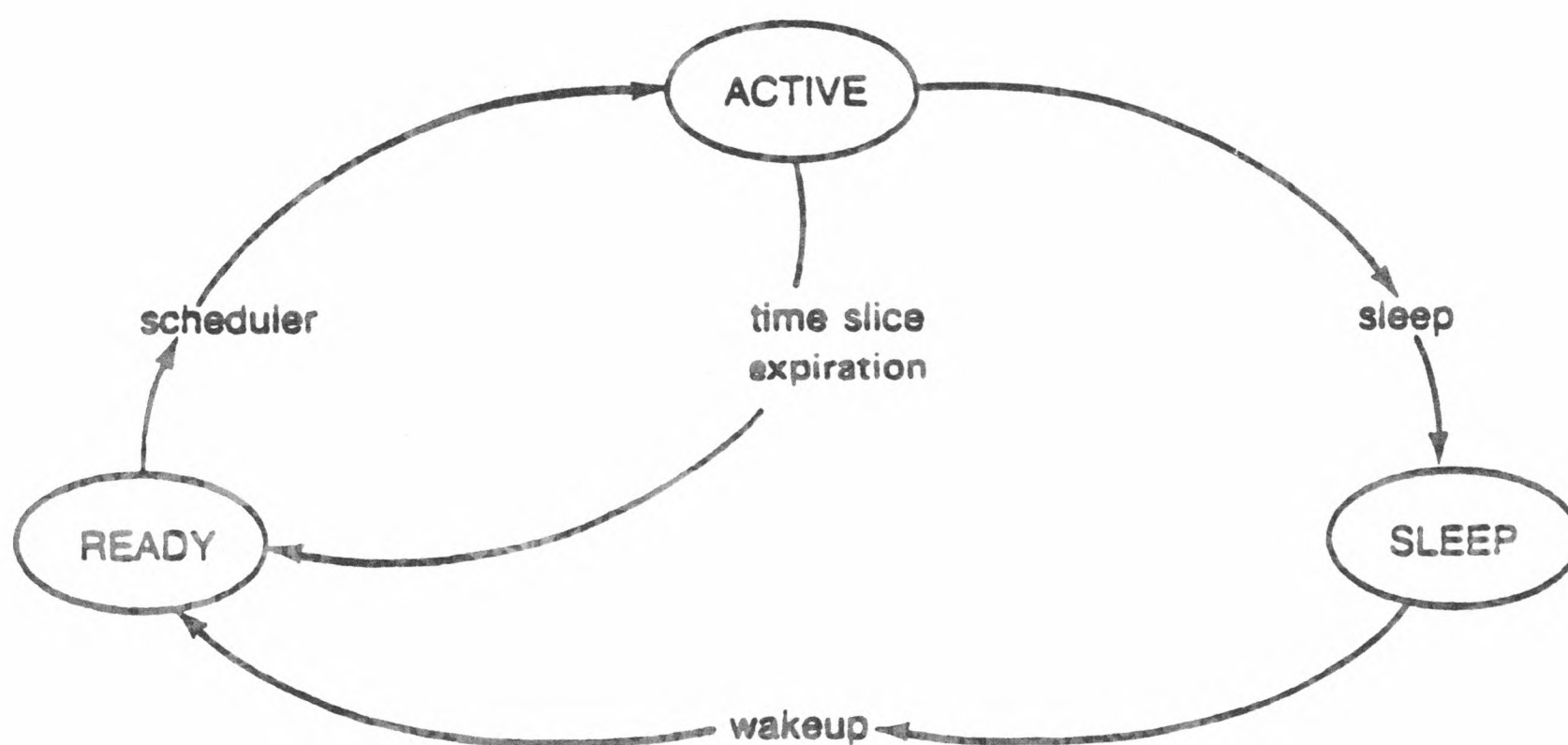


Figure 3. Partial state diagram

A device driver can cause a transition between ACTIVE and SLEEP (when the I/O starts) or SLEEP and READY (when the I/O completes). The sleep function (in the kernel) puts the currently executing process to sleep. The function has this format:

sleep(event, pri)

event

A unique number identifying the reason a process was put to sleep. The process is sleeping for an event. Processes can be asleep waiting for a timer to expire, a terminal key to be pressed, a buffer to be filled, etc. Normally, the event is the address of a data structure; for example, the address of the buffer to be filled.

pri

The priority (relative importance) the process obtains after it "wakes up".

The *wakeup* function activates all processes waiting for a specific event (causes a transition from the SLEEP state to the READY state). The *wakeup* function has this format:

wakeup(event)

event

The event that has occurred.

SLEEP AND WAKEUP

There are four concepts a device driver must consider when it puts processes to sleep and wakes them up:

- The *sleep* can be thought of as a suspension until some condition is met.
- For each *sleep* there has to be a *wakeup*.
- The driver can put more than one process to sleep on the same condition. Therefore, after returning from a sleep, the condition must be examined to be sure that it has been satisfied. If the condition isn't satisfied, the process should be put back to sleep.

For example, two processes could be asleep waiting to receive a message when it is delivered.

When the message is delivered, *wakeup* is called and both processes are moved to the READY list. The first process to execute may receive the message. When the second process finally gets to execute, the driver needs to realize that the message has been received and should put the process back to sleep.

- There is overhead involved in a *wakeup*, so it should only be called if there is a process sleeping.

Device drivers need a method to determine if certain conditions are met when processing a request. For example, the device driver needs to check if a process is sleeping for I/O before issuing a *wakeup*. Device drivers normally use binary flags in a "state variable" to represent required conditions.

Typically, these flags are stored as single bits in an integer. For example:

- *Wakeup* can be called from the top half or the bottom half of the driver.
- *Sleep* can only be called from the top half of the driver.

```
#define CONDITION_X 0x01
#define SLEEP_X 0x02
short state;
```

To set a flag, a binary OR is used to turn on the `CONDITION_X` bit in state:

```
state |= CONDITION_X; /* state = state | CONDITION_X */
```


To test a flag, the binary AND is used:

```
if (state & CONDITION_X)
    /* CONDITION_X is true */
else
    /* CONDITION_X is false */
```

To clear a flag, the binary AND is also used to turn off the bit representing `CONDITION_X`:

```
state &= CONDITION_X;
```

Device drivers generally need flags for at least two conditions:

`CONDITION_X`

Condition X must be satisfied before a processes execution can continue. The process is put to sleep until the condition is satisfied.

NOTE: This condition may not actually be a flag. For example, a process can sleep until a list has a certain number of characters instead of just empty or full.

`SLEEP_X`

One or more processes are sleeping. The routine satisfying `CONDITION_X` only issues a wakeup if `SLEEP_X` is true.

Driver code executed for a process put to sleep includes:

```
while (state & CONDITION_X)    /* while cond. is not satisfied */
{
    state |= SLEEP_X;          /* mark as sleeping */
    sleep(&buffer, PRIORITY);    /* wait for buffer to be filled */
}
copy-buffer-to-user-space;
```

Driver code executed to move a process to the READY state includes:

```
something = goodstuff;    /* the buffer is filled */
state &= CONDITION_X;     /* condition is therefore satisfied */
if (state & SLEEP_X)      /* is anyone sleeping? */
{
    state &= SLEEP_X;      /* no longer sleeping */
    wakeup(&buffer);       /* so wake those waiting */
}
```


Chapter 5.

A Pseudo Device Driver

OVERVIEW

This chapter describes an example device driver, *sd*, that implements the basic functions required by all device drivers

The *sd* device driver is a pseudo character device driver; pseudo means that no hardware is actually being controlled. Each minor device is a mailbox that unrelated processes can use to send or receive messages. Each minor device (*sd0*, *sd1*, ... *sdn*) has the capacity to hold one message.

Figure 1 illustrates process 0 and process 1 communicating through the mailbox represented by minor device *sd0*.

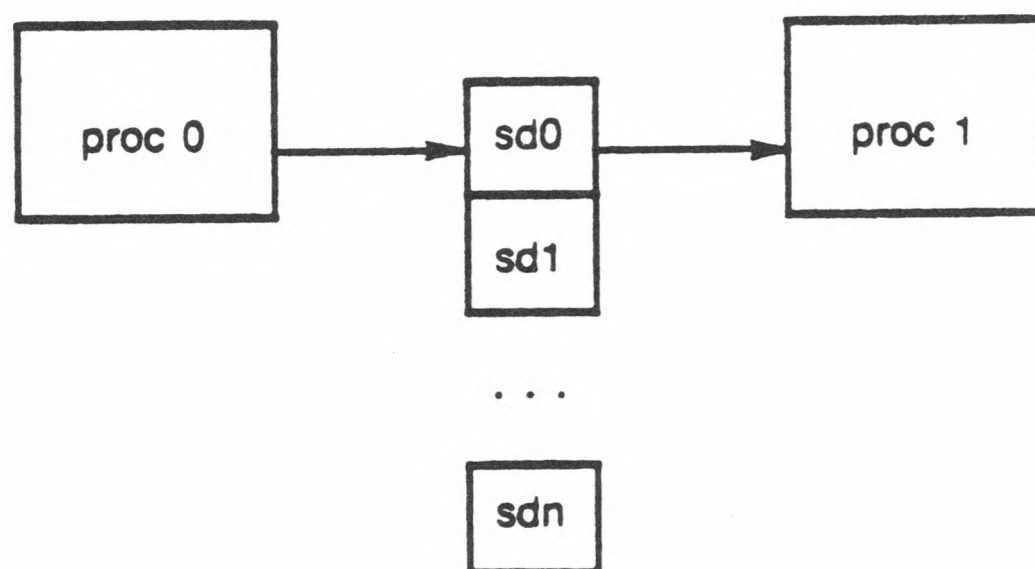


Figure 1. Driver Block Description

Figure 2 shows that Proc 0 communicates with Proc 1 through the *sd* device driver. Requests to the *sd* device driver are made using system calls. The system calls used by the *sd* device driver are:

- *open* - request access to *sd*.
- *close* - finished with *sd*.
- *read* - retrieve a message from the mailbox.
- *write* - store a message in the mailbox.
- *ioctl* - *sd* control command.
 - SDRSTATE - Return (read) the state of the mailbox.
 - SDWSTATE - Set (write) the state of the mailbox.

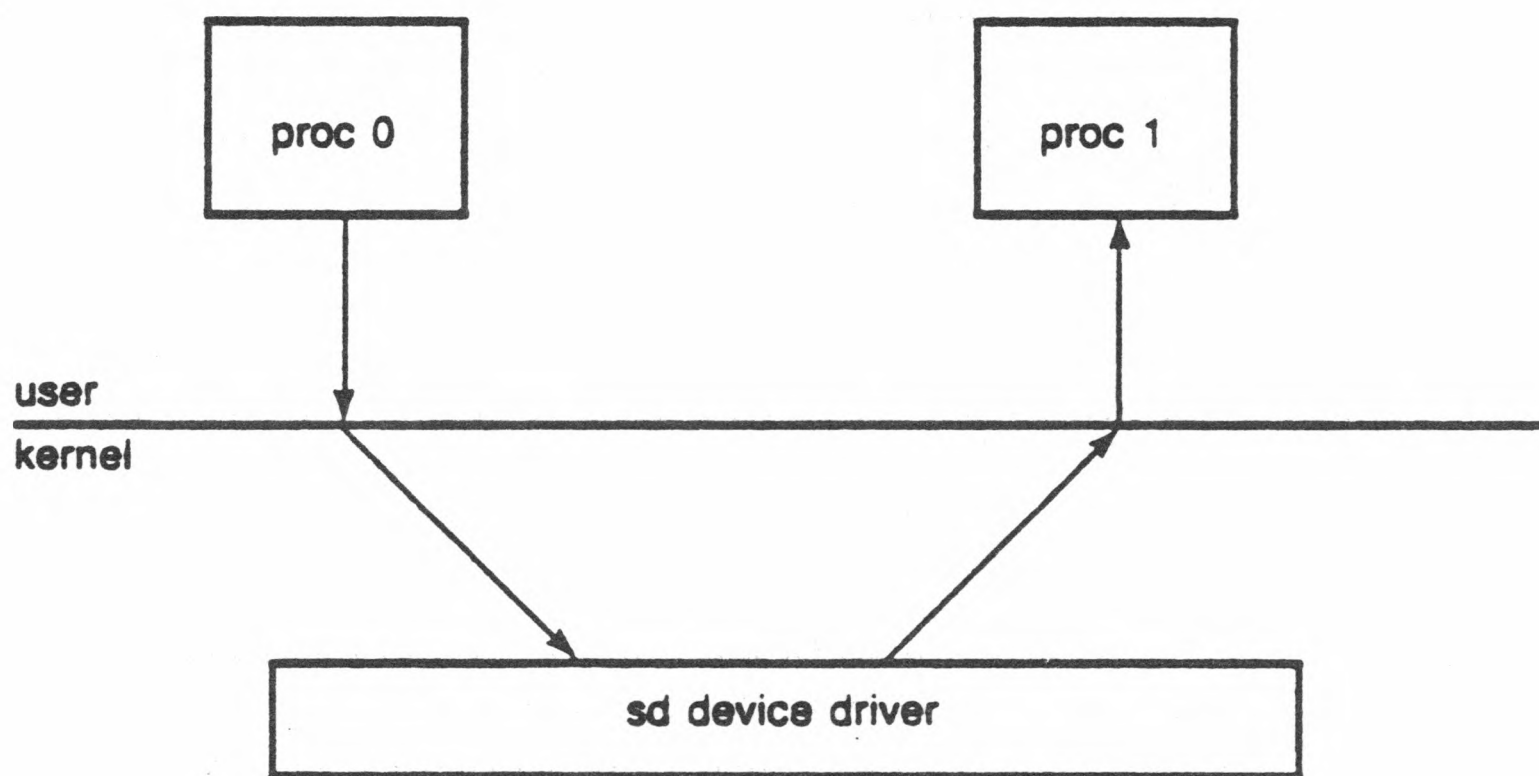


Figure 2. Process Communication

I/O System Control Flow

Figure 3 shows the control flow for the *sd* character device driver. Each system call used by a program corresponds to a call to the device driver. The entry points to the driver are the system call names preceded by the handler prefix for the device. The handler prefix is a short acronym that uniquely identifies each driver.

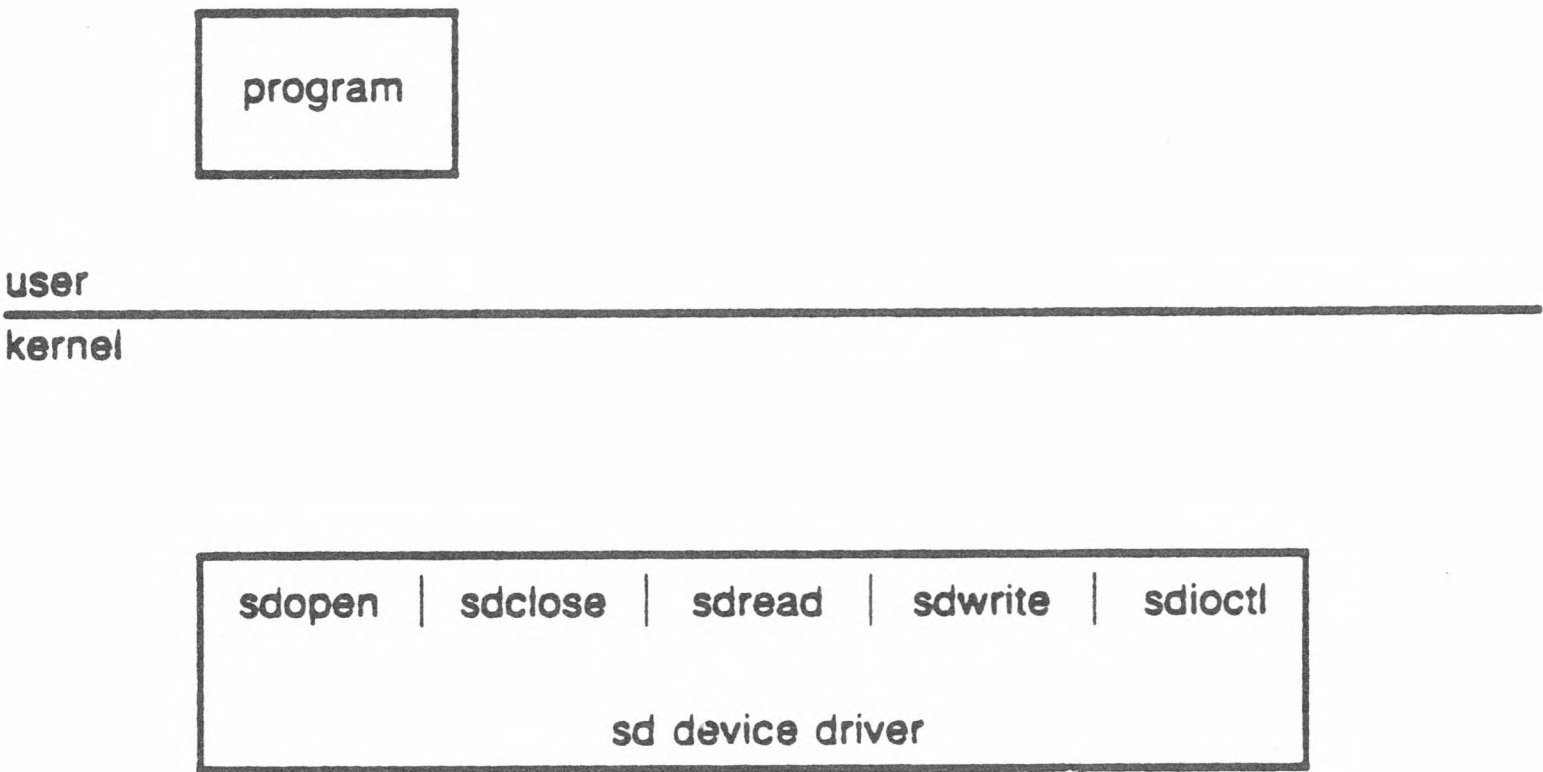


Figure 3. Control Flow

Device Identification

There are many device drivers included in the kernel. The *open(2)* system call identifies which device driver is to be accessed. Because all devices are accessed as if they were files, an *inode* must exist for each device.

An *inode* has a mode field (Figure 4) that contains bit flags. The bit identified by the octal number 020000 is ON if the *inode* represents a block device. The bits identified by 060000 are ON if the *inode* represents a character device. If either of these bits is on, the *inode* is called a "device node".

Device nodes have two numbers instead of block pointers in the *inode*.

- The major device driver number identifies the driver. For example, the HPSIO driver is identified by major device number 11.
- The minor or "sub device" number identifies a specific device. For example, `/dev/tty01` has the minor device number 1, `/dev/tty02` has the minor device number 2, etc.

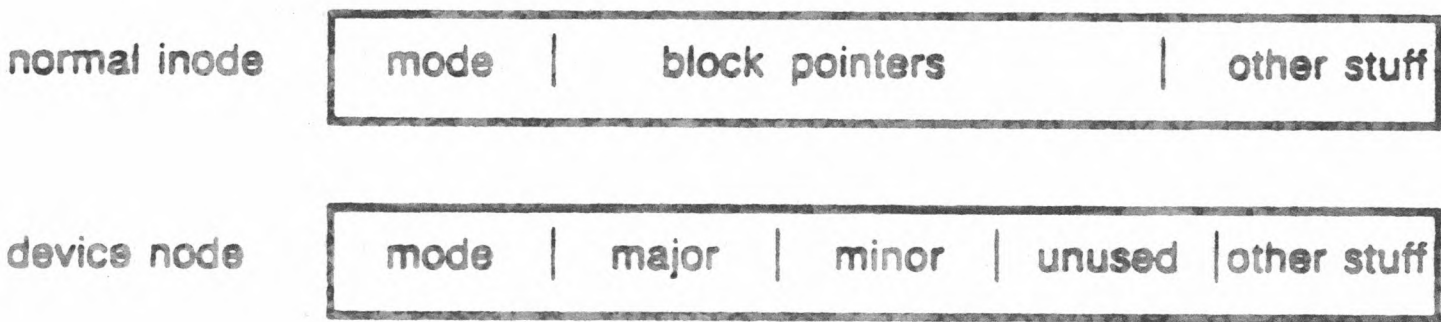


Figure 4. Device Node

Device Creation - `/etc/mknod`

Device nodes are created by the command `/etc/mknod`. The format of the command is:

```
/etc/mknod name c | b major minor
```

The names typically reside in the `/dev` directory. The type of device is identified by `c` (character) or `b` (block). For example:

```
# /etc/mknod /dev/sd c 18 0
```

The character device node `/dev/sd` has major number 18 and minor number 0. Major number 18 was used because 18 is not used for any other device driver in the kernel. The `ls -l` command displays major and minor information for device nodes.

```
$ ls -l /dev/sd
crw-rw-rw-  1 root      18,  0   Aug 20 16:15 /dev/sd
```

Diagram illustrating the output of the `ls -l /dev/sd` command and the meaning of the fields:

- `crw-rw-rw-`: character special file (device node)
- `18,`: major device number
- `0`: minor device number

Figure 5. Device Node Display

Character Device Switch Table

The character device switch table (`cdevsw`) contained in the kernel defines the location of all character

device driver entry points. This table is indexed by the major device number. The *cdevsw* table is defined in the file */usr/include/sys/conf.h*.

```
/*
 * Character device switch.
 */
struct cdevsw {
    int (*d_open)();
    int (*d_close)();
    int (*d_read)();
    int (*d_write)();
    int (*d_ioctl)();
    struct tty *d_ttys;
};

extern struct cdevsw cdevsw[];
```

The table values are defined in the file *conf.c*.

```
struct cdevsw cdevsw[] = {
/* 0*/ necopen,  necclose,  necread,  necwrite,  necioctl,nec_tty,
/* 1*/ nulldev,  nulldev,  mmread, mmwrite,  nodev, 0,
/* 2*/ conopen,  nulldev,  conread,  conwrite,  conioctl,0,
/* 3*/ nodev, nodev, nodev, nodev, nodev, 0,
/* 4*/ syopen, nulldev,  syread, sywrite,  syioctl,0,
/* 5*/ mpcopen,  mpcclose,  mpcread,  mpcwrite,  mpcioc1,0,
/* 6*/ nodev, nodev, nodev, nodev, nodev, 0,
/* 7*/ wdopen, wdclose,  wdread, wdwrite,  wdioc1,0,
/* 8*/ tpopen, tpclose,  tpread, tpwrite,  tpioct1,0,
/* 9*/ nodev, nodev, nodev, nodev, nodev, 0,
/*10*/ hpmopen,  nulldev,  nodev, hpmwrite,  hpmioc1,0,
/*11*/ hpopen, hpclose,  hpread, hpwrite,  hpioct1, hp_tty,
/*12*/ nodev, nodev, nodev, nodev, nodev, 0,
/*13*/ nodev, nodev, nodev, nodev, nodev, 0,
/*14*/ so_open,  so_close,  so_read,  so_write,  so_ioct1,0,
/*15*/ nt_open,  nt_close,  nt_read,  nt_write,  nt_ioct1,0,
/*16*/ nodev, nodev, nodev, nodev, nodev, 0,
/*17*/ nodev, nodev, nodev, nodev, nodev, 0,
/*18*/ sdopen, sdclose,  sdread, sdwrite,  sdioct1,0,
/*19*/ nodev, nodev, nodev, nodev, nodev, 0,
/*20*/ dtopen, dtclose,  dtread, dtwrite,  dtioct1, dt_tty,
/*21*/ nulldev,  nulldev,  cdtread,  cdtwrite,  cdtioct1,0,
/*22*/ nodev, nodev, nodev, nodev, nodev, 0,
/*23*/ nodev, nodev, nodev, nodev, nodev, 0,
/*24*/ nodev, nodev, nodev, nodev, nodev, 0,
/*25*/ erropen,  errclose,  errread,  nodev, nodev, 0,
/*26*/ nodev, nodev, nodev, nodev, nodev, 0,
/*27*/ nulldev,  nulldev,  nodev, nodev, nodev, 0,
};
```

NOTE: Any line entry in the *conf.c* file with all *nodev* and 0 is available for use. *Nodev* mean no device driver is configured with that major number. *Nulldev* means a device driver is configured but has no entry point.

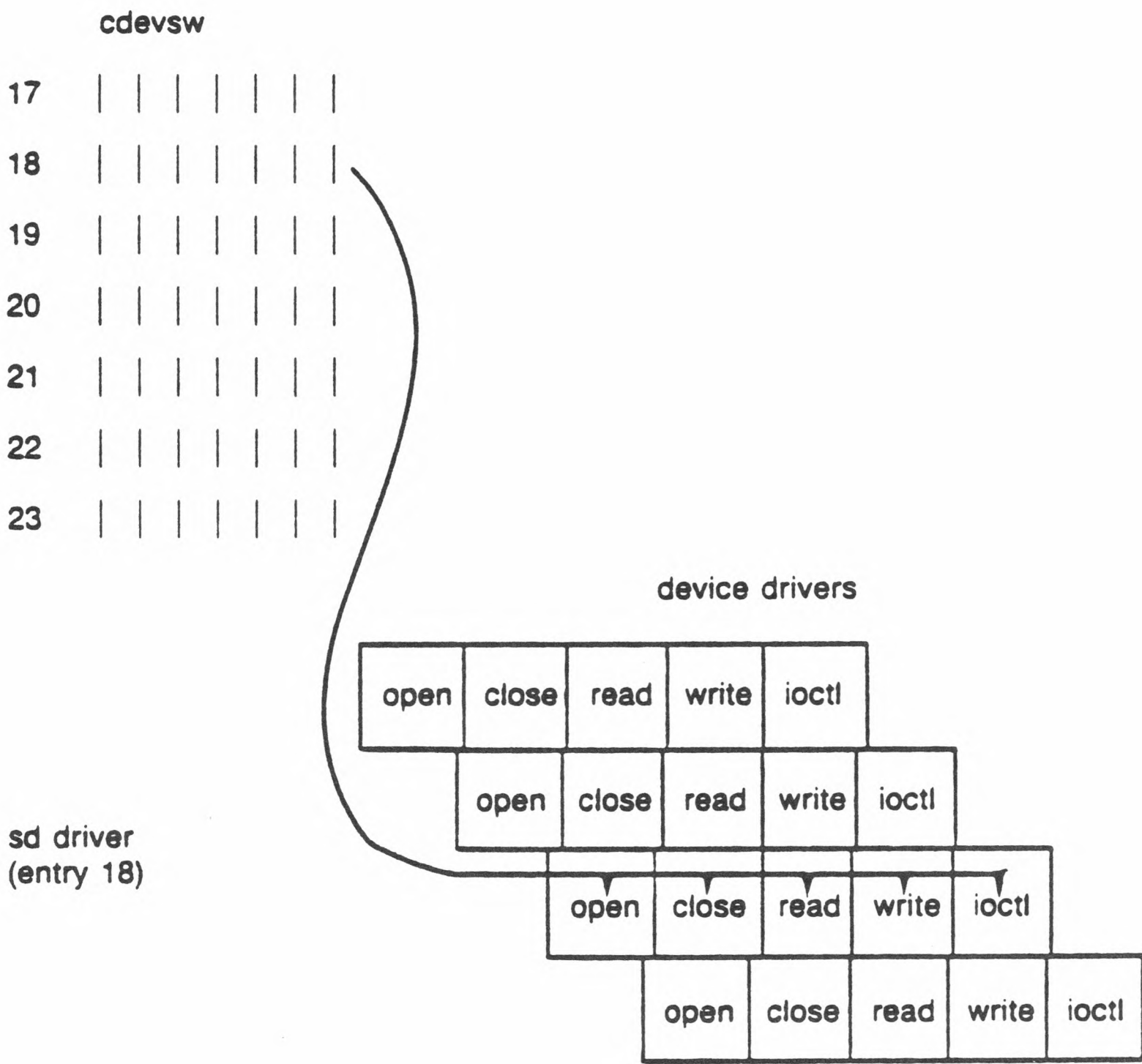


Figure 6. The cdevsw entries

System Include Files

The kernel has many internal structures and constants that can be used by a device driver. These items are defined in include files in the directory */usr/include/sys*. Here are a few of the most often-used include files:

- *types.h* - basic data types such as *uint* for unsigned int and *dev_t* for device type.
- *param.h* - basic kernel parameters such as table sizes
- *sysmacros.h* - C preprocessor macros such as *major(d)* and *minor(d)*
- *buf.h* - buffer and queue header structures.
- *iobuf.h* - queue header structures used by block device drivers.
- *user.h* - user structure definition.
- *proc.h* - process table definition.
- *signal.h* - structures used to send software signals to processes.
- *errno.h* - standard errors defined in the *Programmer Reference*, UP-11762.
- *dir.h* - directory structure.

NOTE: The include files must be specified for the driver.

Sd - Declarations

The primary variables and structures are:

- *sd_cnt*:

extern int sd_cnt

Shows the number of minor devices permitted. This number is a configuration parameter independent of the device driver.

- *sd_sinfo*:

```
extern struct sinfo sd_sinfo[]
```

Defines a mailbox and contains a mailbox for each minor device. The length of *sd-sinfo* array is *sd_cnt*. Refer to Figure 7.

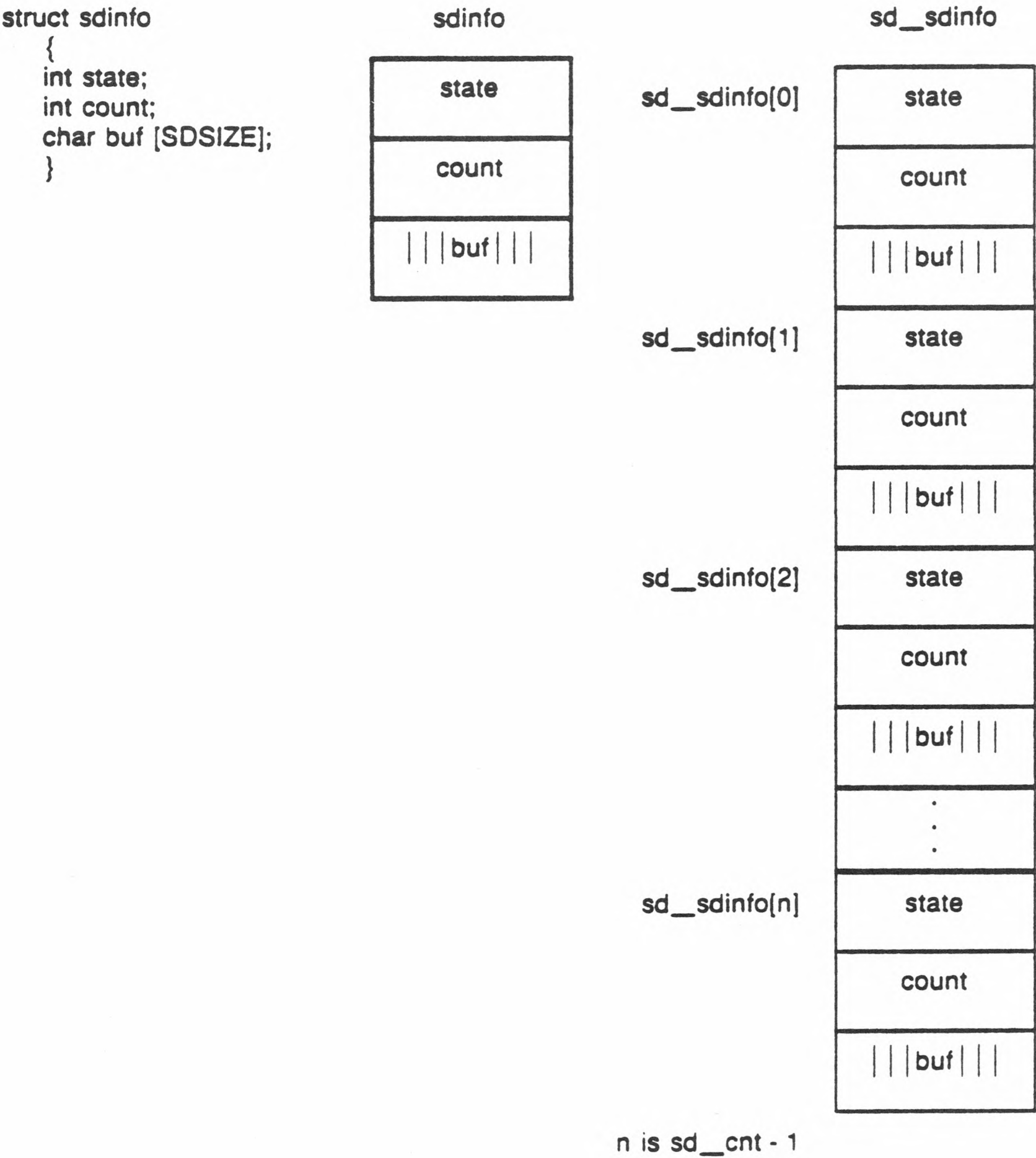


Figure 7. Mailbox Structure

The structure *sd_sinfo* and the variable *sd_cnt* are both defined in the file *conf.c* created by *config (1M)* during the the kernel generation process.

Device Driver Open and Close Routine

The device driver *open* and *close* routines are defined as follows:

```
sdopen(minor_d, flag)
dev_t  minor_d;      /* minor device number */
int flag;             /* the 4 least sig. bits are defined
                       in file.h */

sdclose(minor_d)
dev_t  minor_d;      /* minor device number */
```

The device driver *open* and *close* routines are called using the *cdevsw* table. The minor device number is passed to these routines to identify the *sub* device.

The flag in *sdopen* comes from the value given in the application program (system) *open* call. For example:

```
main()
{
    open("/dev/sd", 2);
}
```

The value of flag comes from the second argument (value is 2). The major number identifies the device driver and is not passed as a parameter.

Sd Definition - sdopen

```
int sdopen(minor_d, flag);  
dev_t minor_d;  
int flag;
```

WHERE

minor_d

minor device number.

flag

defines open mode as in */usr/include/sys/file.h*

- FREAD-read only access was requested
- FWRITE-write only access was requested
- FNDELAY-non blocking
- FAPPEND-all writes will append to end of file (device)

DESCRIPTION

The minor device number is validated. If the system call *open("/dev/sd0", 2)* is made, the value of flag is 3. The argument of 2 in the *open* system call represents *O_RDWR* (See */usr/include/fcntl.h*.) The flag value comes from the values for FREAD and FWRITE in */usr/include/sys/file.h*.

FAILURE CONDITIONS

Invalid device [ENXIO] minor number is out of range.

Related Include Files

The file *fcntl.h* is used within the program. The file *file.h* is used by the device driver. The definition of most flags is the same.

NOTE: In *file.h*, FREAD and FWRITE are bit flags. In *fcntl.h* O_RDONLY, O_WRONLY, and O_RDWR are used. All of the O_ flags in *fcntl.h* can be used by an application and interpreted by the driver and don't affect the driver independent kernel.

```
/usr/include/fcntl.h
```

```
/* fcntl.h 4.2 83/09/25 */
```

```
/*
```

```
 * Flag values accessible to open(2) and fcntl(2)
```

```
 * (The first three can only be set by open)
```

```
*/
```

```
#define O_RDONLY 0
```

```
#define O_WRONLY 1
```

```
#define O_RDWR 2
```

```
#define O_NDELAY FNDELAY /* Non-blocking I/O */
```

```
#define O_APPEND FAPPEND /* append (writes guaranteed at the end) */
```

```
#ifndef F_DUPFD
```

```
/* fcntl(2) requests */
```

```
#define F_DUPFD 0 /* Duplicate fildes */
```

```
#define F_GETFD 1 /* Get fildes flags */
```

```
#define F_SETFD 2 /* Set fildes flags */
```

```
#define F_GETFL 3 /* Get file flags */
```

```
#define F_SETFL 4 /* Set file flags */
```

```
#define F_GETOWN 5 /* Get owner */
```

```
#define F_SETOWN 6 /* Set owner */
```

```
/* flags for F_GETFL, F_SETFL-- copied from <sys/file.h> */
```

```
#define FNDELAY 00004 /* non-blocking reads */
```

```
#define FAPPEND 00010 /* append on each write */
```

```
#define FASYNC 00100 /* signal pgrp when data ready */
```

```
#endif
```

```
/usr/include/sys/file.h
```

```

/* file.h 6.2 83/09/23 */

#ifdef KERNEL
/*
 * Descriptor table entry.
 * One for each kernel object.
 */
struct file {
    int f_flag; /* see below */
    short f_type; /* descriptor type */
    short f_count; /* reference count */
    short f_msgcount; /* references from message queue */
    struct fileops {
        int (*fo_rw)();
        int (*fo_ioctl)();
        int (*fo_select)();
        int (*fo_close)();
    } *f_ops;
    caddr_t f_data; /* inode */
    off_t f_offset;
};

struct file *file, *fileNFILE;
int nfile;
struct file *getf();
struct file *falloc();
#endif

/*
 * flags- also for fcntl call.
 */
#define FOPEN (-1)
#define FREAD 00001 /* descriptor read/receive'able */
#define FWRITE 00002 /* descriptor write/send'able */
#ifdef F_DUPFD
#define FNDELAY 00004 /* no delay */
#define FAPPEND 00010 /* append on each write */
#endif
#define FMARK 00020 /* mark during gc() */
#define FDEFER 00040 /* defer for next gc pass */
#ifdef F_DUPFD
#define FASYNC 00100 /* signal pgrp when data ready */
#endif
#define FSHLOCK 00200 /* shared lock present */
#define FEXLOCK 00400 /* exclusive lock present */

/* bits to save after open */
#define FMASK 00113
#define FCNTLCANT (FREAD|FWRITE|FMARK|FDEFER|FSHLOCK|FEXLOCK)

```

```

/* open only modes */
#define FCREAT      01000      /* create if nonexistant */
#define FTRUNC      02000      /* truncate to zero length */
#define FEXCL       04000      /* error if already created */

#ifndef F_DUPFD
/* fcntl(2) requests--from <fcntl.h> */
#define F_DUPFD 0 /* Duplicate fildes */
#define F_GETFD 1 /* Get fildes flags */
#define F_SETFD 2 /* Set fildes flags */
#define F_GETFL 3 /* Get file flags */
#define F_SETFL 4 /* Set file flags */
#define F_GETOWN 5 /* Get owner */
#define F_SETOWN 6 /* Set owner */
#endif

/*
 * User definitions.
 */

/*
 * Open call.
 */
#define O_RDONLY 000 /* open for reading */
#define O_WRONLY 001 /* open for writing */
#define O_RDWR 002 /* open for read & write */
#define O_NDELAY FNDELAY /* non-blocking open */
#define O_APPEND FAPPEND /* append on each write */
#define O_CREAT FCREAT /* open with file create */
#define O_TRUNC FTRUNC /* open with truncation */
#define O_EXCL FEXCL /* error on create if file exists */

/*
 * Flock call.
 */
#define LOCK_SH 1 /* shared lock */
#define LOCK_EX 2 /* exclusive lock */
#define LOCK_NB 4 /* don't block when locking */
#define LOCK_UN 8 /* unlock */

/*
 * Access call.
 */
#define F_OK 0 /* does file exist */
#define X_OK 1 /* is it executable by caller */
#define W_OK 2 /* writable by caller */
#define R_OK 4 /* readable by caller */

/*
 * Lseek call.

```



```

*/
#define L_SET      0 /*absolute offset*/
#define L_INCR     1 /*relative to current offset*/
#define L_XTND     2 /*relative to end of file*/

#ifdef KERNEL
#define GETF(fp, fd) {
    if ((unsigned)(fd) >= NOFILE || ((fp) = u.u_ofile[fd] == NULL) {
        u.u_error = FBADF;
        return;
    }
}
#define DTYPE_INODE 1 /*file*/
#define DTYPE_SOCKET 2 /*communications endpoint*/
#endif

```

Sdopen Listing

```

/*-----
 * sdopen -- validate the minor device number.
 *-----
 */
void sdopen(minor_d, flag)
    dev_t    minor_d; /* minor device number */
    int      flag;    /* flag indicating read/write */
{
    if (minor_d >= sd_cnt)
    {
        u.u_error = ENXIO;
        return;
    }
}

```

Sdopen is called using the *cdevsw* table and validates the minor number.

Sd Definition - sdclose

```

int sdclose(minor_d);
dev_t minor_d;

```

WHERE

minor_d
minor device number.

DESCRIPTION

Does nothing. The driver's *close* routine is only called on the last close to an open device. If two processes have a device open simultaneously, the *close* routine is only called after both processes call *close*.

FAILURE CONDITIONS

None

Sdclose Listing

```
/*-----  
* sdclose -- not much  
*-----  
*/  
void sdclose(minor_d)  
    dev_t minor_d;    /* minor device number */  
  
    {  
        printf("sdclose\n");  
    }
```

Accessing User Memory Space (ioctl Functions)

Programs can specify invalid addresses when making requests from the kernel. To protect the kernel from program errors, data is NEVER copied directly from kernel to user space.

Data can be copied from kernel space to user space using special kernel functions. Two of these functions are *suword(k)* and *fuword(k)*.

suword

Set user word. Transfer one word (four bytes) from kernel space to user space. If *suword* fails because of a bad user address, the value -1 is returned.

fuword

Fetch user word. Transfer one word (four bytes) from user space to kernel space. The return value of -1 indicates an error.

Examples:

```
if (suword(wordptr, 12) == -1)    /* move 12 to users buffer */
    { ... error ... }
```

```
if ((val = fuword(wordptr) == -1) /* retrieve an integer */
    { ... error ... }
```

For more information on accessing user space, see: *fubyte* (k), *fuword* (k), *fustr* (k), *subyte* (k), *sustr* (k), *copyin* (k), *copyout* (k), *iomove* (k).

Sd Definition - *sdioctl*

```
int sdioctl(minor_d, cmd, arg);
dev_t minor_d;
int cmd;
int * arg;
```

WHERE

minor_d

minor device number.

cmd

command or function to perform.

arg

pointer to argument defining a parameter block

DESCRIPTION

These commands are available:

SDRSTATE

Copy state of mailbox to the user's argument.

SDWSTATE

Copy user's argument to the state of the mailbox.

FAILURE CONDITIONS

EINVAL is returned if an invalid command is requested. Bus errors or addressing errors are possible if arg is not a valid address within user space.

Example Use of SDIOCTL

```
#include <fcntl.h>
#include </usr/acct/yours/sd/sd.h>
main()
{
    int fd;    /* file descriptor */
    int state; /* result of sdiocctl */

    fd = open("/dev/sd1", O_RDWR);
    ioctl(fd, SDRSTATE, &state);
    printf("value of state variable is %x n", state);
}
```

NOTE: The driver manipulates the variables associated with *sd_sdinfo* (see *sdiocctl*).

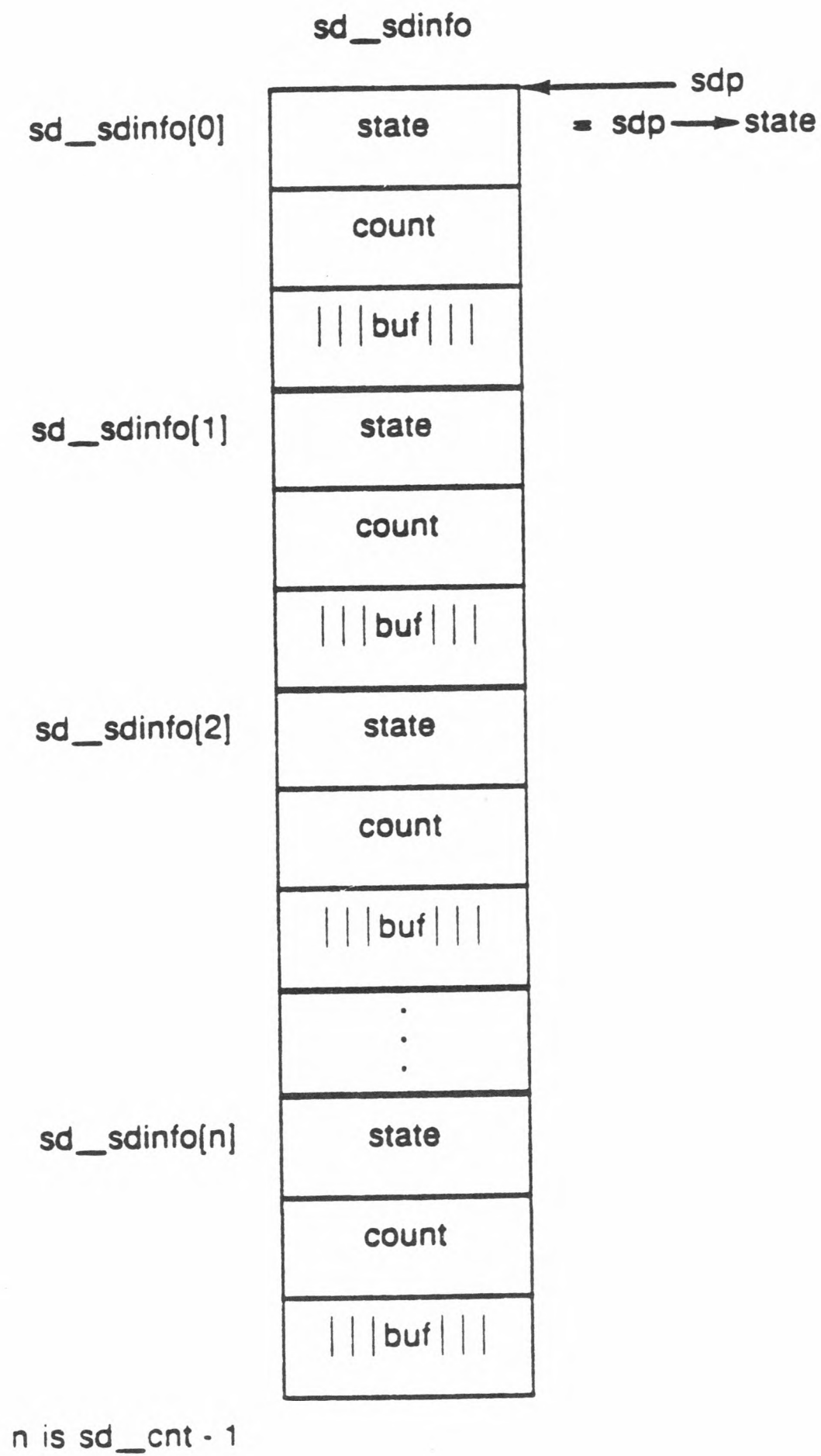


Figure 8. Functions of ioctl

Sdioctl Listing

```
/*-----  
* sdioc1 -- perform the control functions  
*-----  
*/  
void sdioc1(minor_d, cmd, arg)  
    dev_t minor_d;  
    int cmd;  
    int * arg;  
  
    {  
        struct sinfo * sdp;  
  
        sdp = &sd_sinfo[minor_d];  
        switch (cmd)  
        {  
            case(SDRSTATE):  
                suword(arg, sdp->state);  
                break;  
            case(SDWSTATE):  
                sdp->state = fuword(arg);  
                break;  
            default:  
                u.u_error = EINVAL;  
                return;  
        }  
    }
```

PROCESS MANAGEMENT, U STRUCTURE

Information needed when a process is executing is stored in the System Data Segment. This data segment is a structure user as defined in the file */usr/include/sys/user.h*.

There is a user structure for every process. Only the user structure for the currently executing process is mapped into kernel space. When executing kernel code, the variable *u* is used to access the user structure. An interrupt service routine (ISR) must not use this structure because the ISR cannot predict which process is going to be active when the interrupt occurs.

Some *u* structure fields used for communicating with a device driver are:

u.u_base
address of user's buffer

u.u_count
number of bytes to be transferred between user and kernel space.

u.u_offset
byte offset into the file or device to be accessed.

u.u_error
set by the driver to a non zero value to indicate an error.

If *u.u_error* is set to a non-zero value, a -1 is returned from the system call to the user program. If *u.u_error* is not set, a zero is returned from the *open*, *close*, *seek*, and *ioctl* function calls. The number of bytes read is returned from the *read* and *write* system calls. For example, if a user program makes a read system call as follows:

```
main()  
{  
    read (fd, buf, 10);  
}
```

The device driver is called with the following values in the *u* structure set:

```
u.u_base = buf;  
u.u_count = 10;
```

The device driver may have code as follows:

```
sdread(minor_d)  
dev_t minor_d; /* minor device number */  
{
```

```
    for(; u.u_count; --u.u_count, ++u.u_base)
    {
        c = read_device();
        if (c == -1)
            break;
        subyte(u.u_base, c);
    }

    ...
}
```

If the device driver read routine changes the value of *u.u_count* to 2, the value returned from the user's read call is 8.

Data Movement and the U Structure (Passc Cpass)

Device drivers may need to move a string of characters to user space. The function call *subyte(u_base, string[i])* copies one byte from kernel space to user space. The following fields in the *u* structure needs to be updated after *subyte* is used:

<i>u_count</i>	decremented
<i>u_offset</i>	incremented
<i>u_base</i>	incremented

A single function *passc(c)* moves a character into user space and adjusts the three fields in the *u* structure. *Passc(c)* returns -1 when the last character the user requested has been copied. For example, when *u_count* is decremented to 0. *Passc(c)* also returns -1 if an error occurs and sets *u_error* = EFAULT.

The function *cpass()* returns a byte fetched from user space and updates the same three parameters.

Cpass() returns -1 when *u_count* signals that all characters have been read (*u_count* is 0).

Sd Definition - sdread

```
int sdread(minor_d);  
dev_t minor_d;
```

WHERE

minor_d
is the minor device number.

DESCRIPTION

Attempts to read a message from the mailbox. If there is no message, sleeps until there is a message. The flag `SDMESSAGE` signals there is a message to read. The flag `SDRSLP` signals that the process that called *read* is asleep. The flag `SDWSLP` signals that the process that called *write* is asleep. After the message has been read, wakeup any processes that are sleeping, (waiting for the mailbox to become empty.)

FAILURE CONDITIONS

None

Sdread Listing

```
/*-----  
* sdread -- read a message from the buffer.  
* sleep if there is no message in the buffer  
* transfer the message to user space  
* wakeup any write processes waiting for an empty buffer  
*-----  
*/  
void sdread(minor_d)  
    dev_t minor_d;  
    {  
        struct sinfo * sdp;    /* pointer to minor device info structure */  
        char * strp;           /* pointer to message in buf */  
  
        sdp = &sd_sinfo[minor_d];  
        strp = sdp->buf;  
  
        while (!(sdp->state & SDMESSAGE)) /* sleep if no message in buf */  
            {  
                sdp->state |= SDRSLP;    /* mark as sleeping */  
                sleep(sdrslp(sdp), SDPRI); /* sleep on address */  
            }  
  
        while (sdp->count-->0) /* transfer message to user space */  
            if (passc(*strp++) == -1)  
                break;
```

```
sdp->state &= SDMESSAGE;      /* there is no longer a message */
if (sdp->state & SDWSLP)      /* if there are processes sleeping */
{
    sdp->state &= SDWSLP;      /* clear sleep state */
    wakeup(sdws1p(sdp));      /* wakeup all processes sleeping */
}
```

Sd Definition - sdwrite

```
int sdwrite(minor_d);
dev_t minor_d;
```

WHERE *minor_d*
is the minor device number.

DESCRIPTION

Attempts to write a message to the mailbox. If there is a message in the mailbox, sleeps until the message is removed. Once the message has been written, wakeup any processes that are sleeping (waiting for the mailbox to become full).

FAILURE CONDITIONS

None

Sdwrite Listing

```
/*-----
 * sdwrite -- write a message into the buffer.
 * sleep if there is a message in the buffer
 * transfer the message from user space
 * wakeup any read processes waiting for an empty buffer
 *-----
 */
void sdwrite(minor_d)
    int minor_d;
```

```
{
struct sinfo * sdp;    /* pointer to minor device info structure */
char * strp;           /* pointer to message in buf */
int i;                 /* counter */
int c;                 /* character moved from user space */

sdp = &sd_sinfo[minor_d];
strp = sdp->buf;

while (sdp->state & SDMESSAGE)    /* sleep if message in buf */
{
    sdp->state != SDWSLP;          /* mark as sleeping */
    sleep(sdslp(sdp), SDPRI);      /* sleep on address */
}

sdp->count = 0;
for (i=SDSIZE; i--; sdp->count++) /* transfer message from user space */
    if((c = cpass()) == -1)
        break;
    else
        *strp++ = c;

sdp->state != SDMESSAGE;          /* there is a message in the buffer */
if (sdp->state & SDRSLP)           /* if there are processes sleeping */
{
    sdp->state &= SDWSLP;          /* clear sleep state */
    wakeup(sdrslp(sdp));          /* wakeup all processes sleeping */
}
}
```

CREATING THE NEW KERNEL

The following lines in the *master* file describe the *sd* driver:

*1	2	3	4	5	6	7mb	8mc	9	10
sd	0	37	4	sd	0	0	18	8	0

The following line in the *dfile* for the *config* program describes the *sd* driver:

sd	0	0	0
----	---	---	---

The makefile listed in the previous chapter can be used to create the kernel.

These commands create the device nodes:

```
# /etc/mknod /dev/sd0 c 18 0
# /etc/mknod /dev/sd1 c 18 1
# /etc/mknod /dev/sd2 c 18 2
# /etc/mknod /dev/sd3 c 18 3
# /etc/mknod /dev/sd4 c 18 4
# /etc/mknod /dev/sd5 c 18 5
# /etc/mknod /dev/sd6 c 18 6
# /etc/mknod /dev/sd7 c 18 7
```

This program writes "hi" to mailbox */dev/sd0*:

```
main()
{
    int fd;
    fd = open("/dev/sd0", 2);
    write(fd, "hi", 2);
}
```

The following program reads from mailbox */dev/sd0* and prints the results to standard out:

```
main()
{
    int fd, cnt;
    char buf[4];
    fd = open("/dev/sd0", 2);
    cnt = read(fd, buf, 3);
    buf[cnt]=0;
    printf(buf);
}
```

Sd LISTING

The *sd* driver is separated into many files; however, these files are not separately compiled modules. The following is the code for the *sd* driver. Each segment of the code is described in the previous sections.

```
/*-----
 * sd.c -- simple device driver.
 * this pseudo device driver implements a queue of characters
 * between two unrelated processes. The concept is similar to
 * a named pipe. There is a read side of the device and a write
 * side of the device.
 *-----
 */
#include <sys/types.h>
#include <sys/param.h>
#include <sys/signal.h>
#include <sys/errno.h>
#include <sys/dir.h>
#include <sys/user.h>
#include <sys/sd.h>
#include <fcntl.h>
#include <sd.h>

/*-----
 * console messages defined if DEBUG is defined
 *-----
 */
#define DEBUG
#ifdef DEBUG          /* if console debug messages are needed */
# define display(a,b) sddisp(a,b)
# define print2(a,b) printf(a,b)
#else
# define display(a,b)
# define print2(a,b)
#endif

/*-----
 * information defined during configuration
 *-----
 */
extern struct sdinfo sd_sdinfo[]; /* information for each minor device */
extern int sd_cnt;                /* number of minor devices */
/*-----
 * sd.h -- include file for sd driver
 *-----
 */
```

```

/*-----
 * information structure for each sd minor device
 *-----
 */
#define SDSIZE 3          /* max number of characters in message */
struct sdinfo
{
    int state;            /* state of the minor device */
    int count;            /* number of characters in message */
    char buf[SDSIZE];     /* characters in message */
};

/*-----
 * possible values of state
 *-----
 */
#define SDMESSAGE 0x01 /* there is a message in buf */
#define SDRSLP 0x02    /* a process is asleep in sdread */
#define SDWSLP 0x04    /* a process is asleep in sdwrite */

/*-----
 * software priority to sleep at
 *-----
 */
#define SDPRI PZERO + 5 /* sleep priority, signal interruptible */

/*-----
 * ioctl commands
 *-----
 */
#define SDRSTATE 1      /* read the state variable for the minor device */
#define SDWSTATE 2      /* write the state variable */

/*-----
 * the read processes and the write processes need to sleep for an event.
 * this event is arbitrarily defined as an address in buf
 *-----
 */
#define sdrs1p(sdp) &sdp->buf[0] /* address for read process to sleep */
#define sdws1p(sdp) &sdp->buf[1] /* address for write process to sleep */
/*-----
 * sdopen -- validate the minor device number.
 *-----
 */
void sdopen(minor_d, flag)
    dev_t    minor_d;    /* minor device number */
    int      flag;       /* flag indicating read/write */
{
    if (minor_d >= sd_cnt)

```



```
    {
        u.u_error = ENXIO;
        return;
    }

}

/*-----
 * sdclose -- not much
 *-----
 */
void sdclose(minor_d)
    dev_t minor_d;    /* minor device number */

{
    printf("sdclose\n");
}

/*-----
 * sdioc1 -- perform the control functions
 *-----
 */
void sdioc1(minor_d, cmd, arg)
    dev_t minor_d;
    int cmd;
    int * arg;

{
    struct sdinfo * sdp;

    sdp = &sd_sdinfo[minor_d];
    switch (cmd)
    {
        case(SDRSTATE):
            suword(arg, sdp->state);
            break;
        case(SDWSTATE):
            sdp->state = fuword(arg);
            break;
        default:
            u.u_error = EINVAL;
            return;
    }
}

/*-----
 * sdread -- read a message from the buffer.
 * sleep if there is no message in the buffer
 * transfer the message to user space
 * wakeup any write processes waiting for an empty buffer
 *-----
 */
void sdread(minor_d)
```

```

dev_t minor_d;
{
    struct sinfo * sdp;    /* pointer to minor device info structure */
    char * strp;           /* pointer to message in buf */

    sdp = &sd_sinfo[minor_d];
    strp = sdp->buf;

    while (!(sdp->state & SDMESSAGE)) /* sleep if no message in buf */
    {
        sdp->state |= SDRSLP;          /* mark as sleeping */
        sleep(sdrslp(sdp), SDPRI);     /* sleep on address */
    }

    while (sdp->count-- > 0)            /* transfer message to user space */
        if (passc(*strp++) == -1)
            break;

    sdp->state &= SDMESSAGE;            /* there is no longer a message */
    if (sdp->state & SDWSLP)            /* if there are processes sleeping */
    {
        sdp->state &= SDWSLP;          /* clear sleep state */
        wakeup(sdwsip(sdp));          /* wakeup all processes sleeping */
    }
}

/*-----
 * sdwrite -- write a message into the buffer.
 * sleep if there is a message in the buffer
 * transfer the message from user space
 * wakeup any read processes waiting for an empty buffer
 *-----
 */
void sdwrite(minor_d)
    int minor_d;
{
    struct sinfo * sdp;    /* pointer to minor device info structure */
    char * strp;           /* pointer to message in buf */
    int i;                 /* counter */
    int c;                 /* character moved from user space */

    sdp = &sd_sinfo[minor_d];
    strp = sdp->buf;

    while (sdp->state & SDMESSAGE)     /* sleep if message in buf */
    {
        sdp->state |= SDWSLP;          /* mark as sleeping */
        sleep(sdwsip(sdp), SDPRI);     /* sleep on address */
    }

    sdp->count = 0;

```

Chapter 5

```
for (i=SDSIZE; i--; sdp->count++) /* transfer message from user space */
    if((c = cpass()) == -1)
        break;
    else
        *strp++ = c;

sdp->state != SDMESSAGE;          /* there is a message in the buffer */
if (sdp->state & SDRSLP)          /* if there are processes sleeping */
{
    sdp->state &= SDWSLP;         /* clear sleep state */
    wakeup(sdrslp(sdp));         /* wakeup all processes sleeping */
}
}
```


Hardware Device Control

Chapter 6.

Hardware Device Control

HARDWARE CONTROLLER REGISTERS

Any controller or circuit board that meets *Multibus* specifications can be added to the system. A controller may have the intelligence to control many devices. For example, a terminal controller may control a number of terminals. The goal of the device driver is to provide the interface for these boards.

Controller functions are performed by writing and reading the values of addressable registers that reside on the controller. For example, writing a certain bit pattern to a control register on a terminal controller may change the baud rate of the controller. Writing a value to a data register on a terminal controller may cause the character 'a' to be sent to a terminal. Reading the value of a data register may return a value corresponding to a keystroke. Also, a controller has the ability to interrupt the processor when it has completed a task.

There is no general description of registers that can describe all controllers; each controller is unique. Figure 1 provides a basic block description of a typical controller.

READING AND WRITING DEVICE REGISTERS

The PMC (Processor Memory Controller) contains the MC69010 or MC68020 microprocessor chip and memory management unit. The PMC board executes process code and controls a device by reading and writing to registers located on the controller.

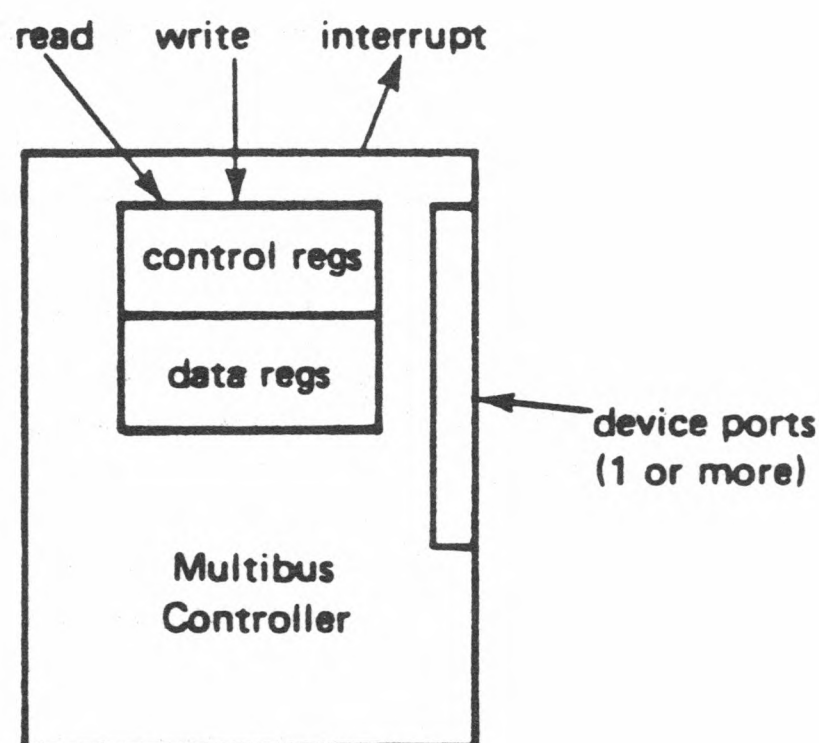


Figure 1. Controller block description

Multibus specifies a special protocol to use when reading and writing device registers. This protocol is called *Multibus I/O*. Kernel code (for example, the device driver) accesses device registers (*Multibus I/O*) when making reference to memory in the 64K range of addresses:

0x1f0000 - 0x1fffff

The actual address bits placed on the *Multibus* are the 16 least significant bits. The controllers respond to *Multibus I/O* addresses in the range:

0x0000 - 0xffff

Only the least significant 4 hex digits are actually placed on the *Multibus I/O* lines. Typically, the addresses the controller responds to are adjustable by firmware or strapping. If, for example, a controller has two 2-byte registers that respond to the *Multibus*

I/O address of 0x0012 and 0x0014, the following C code writes the value 63 to one register:

```
*(short *) (0x1f0012) = 63;
```

and this code reads the value of the other register and stores it in the variable val;

```
short val;  
val = *(short *) (0x1f0014);
```

The standard method of handling the registers on a device is to define them as a structure. The following C declaration defines the device.

```
struct device  
{  
    short data; /* data register */  
    short csr;  /* control status register */  
}  
struct device *dev_addr = (struct device*)0x1f0012;
```

To write to the data register, this C statement is executed:

```
dev_addr->data = 63;
```

To read from the control/status register, this C statement is executed:

```
c = dev_addr->csr;
```

BYTE ORDERING

As shown in Figure 2, processors use one of two different schemes for byte ordering of 16-bit memory words. The 68000 family requires that 16-bit words reside on even byte boundaries and puts the most significant byte in the least significant address.

Non-68000 processors put the most significant byte in the most significant address.

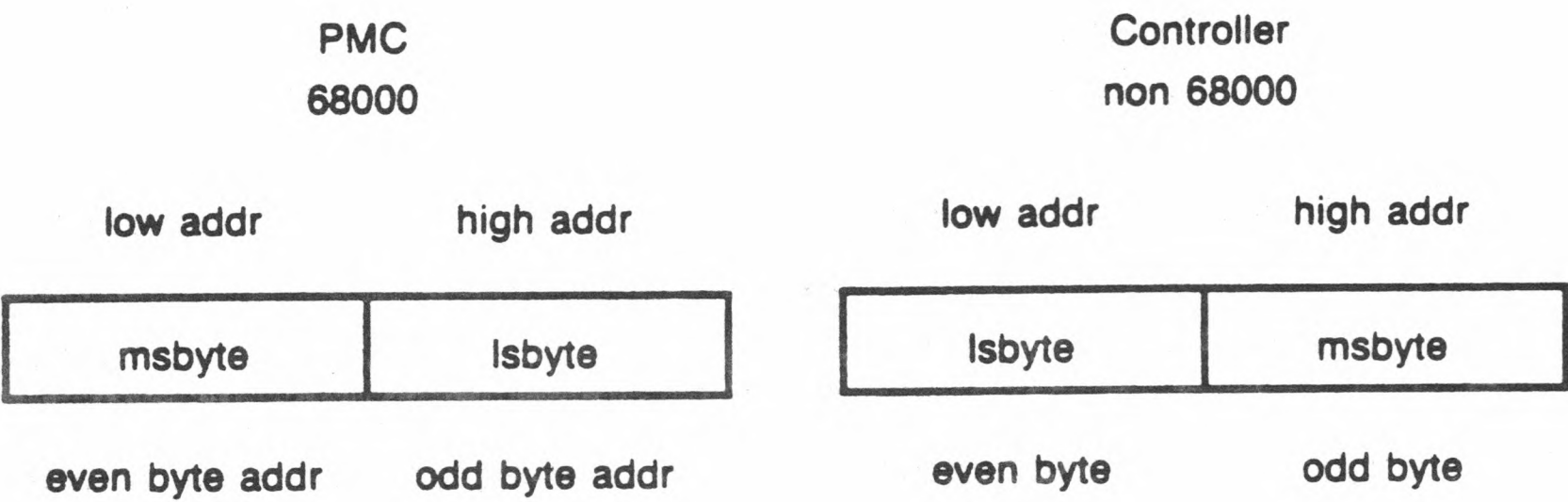


Figure 2. Byte Ordering

It's important to understand exactly how the controller uses *Multibus* address and data lines when performing *Multibus* memory and *Multibus* I/O access. The following section describes how the PMC uses the address and data lines during *Multibus* access.

During word transfers (16-bits) shown in Figure 3, the most significant byte in PMC local system memory corresponds to the most significant *Multibus* data bits (d8 - df). The least significant byte corresponds to the least significant *Multibus* data bits (d0 - d7). Both the local system memory (LSM) address and the *Multibus* address must be even for word transfers.

mov.w 10, 1f0000

PMC

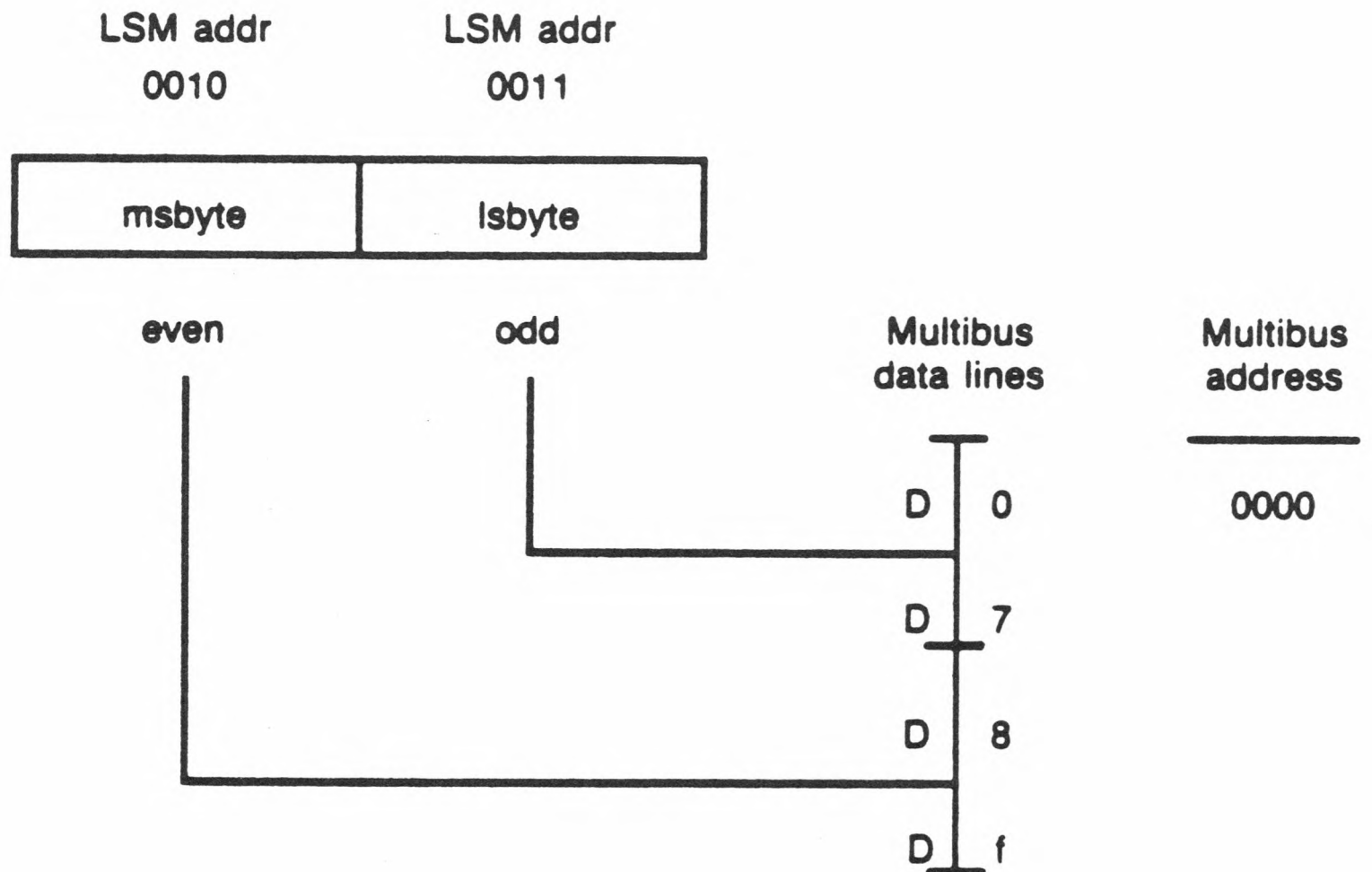


Figure 3. Word Transfers

During byte transfers (8-bits), the byte in PMC local system memory corresponds to the least significant Multibus data bits (d0 - d7). The least significant bit of the address is always toggled. Figures 4 and 5 show the relationship of even and odd byte transfers.


```
mov.b 10,1f0000    PMC
```

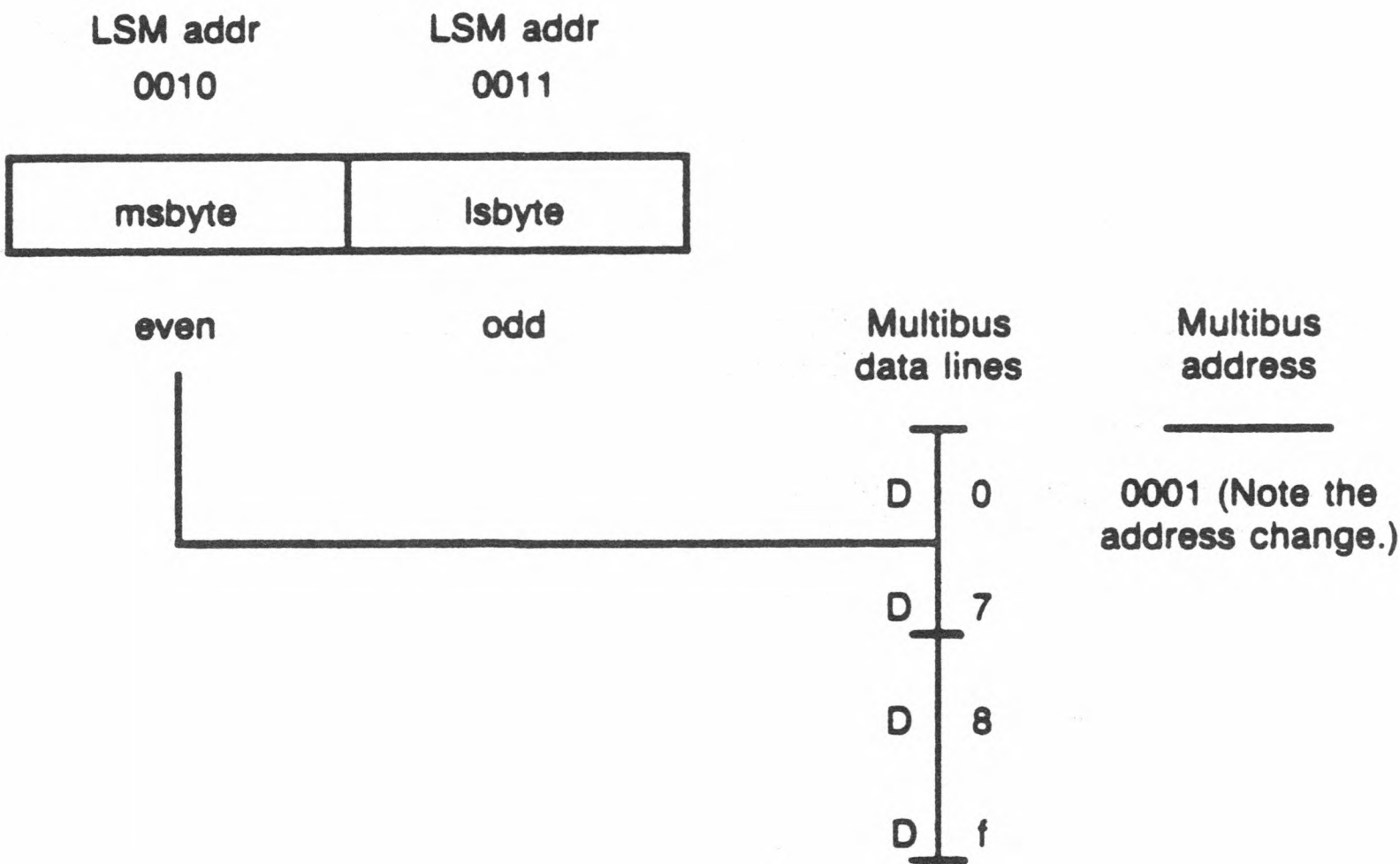


Figure 4. Even Byte Transfers

Bytes in PMC local system memory may have to be swapped before the controller accesses them. The least significant bit of the address may need to be toggled before accessing byte addressable registers.

mov.b 11, 1f0001 PMC

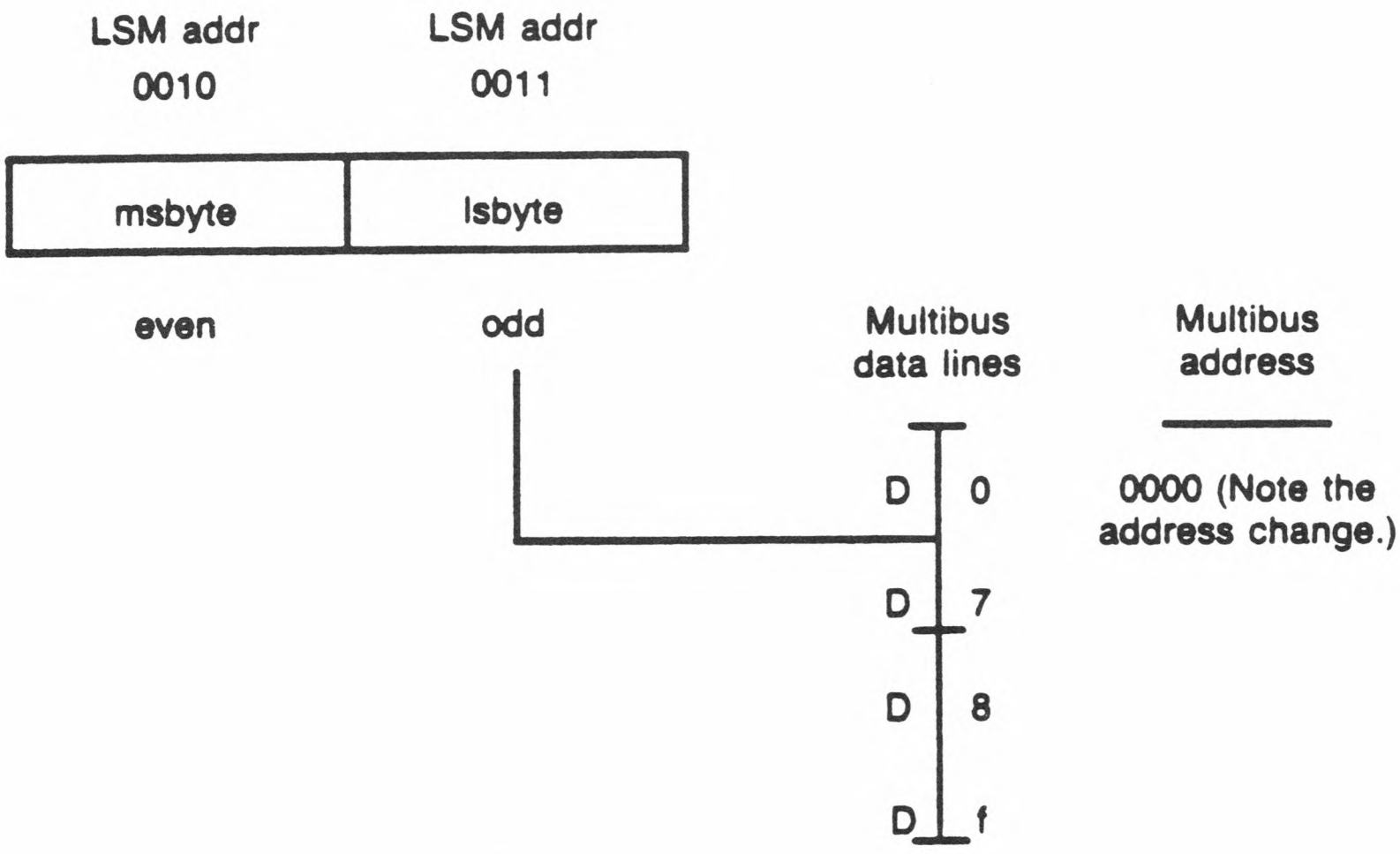


Figure 5. Odd Byte Transfers

MULTIBUS ADDRESS SPACE AVAILABLE TO CUSTOMERS

The physical *Multibus* I/O addresses, 0x8000 - 0x9fff, are reserved exclusively for customer use. Only controllers that recognize addresses in this range can be added to the system. These board addresses correspond to the logical addresses, 0x1f8000 - 0x1f9fff. Other boards must respond to the common bus request (CBRQ/) signal and relinquish bus

control within 10 microseconds.

BUS VECTORED INTERRUPTS

The bus permits two different types of interrupt devices to communicate with the PMC:

- Bus vectored interrupt devices.
- Auto vectored interrupt devices.

NOTE: Only bus vectored devices can be added to the system.

When an external device generates an interrupt, it identifies itself to the PMC by passing a one byte vector to the MC69010 or MC68020. The vector is an index into a vector table (Figure 6).

The contents of this table are the addresses of calls to a common dispatcher. The common dispatcher determines the vector number and indexes a table of interrupt service routines (ISRs) that handle the interrupts. This ISR table is called UNIVec and is created during configuration and stored in the *univec.c* file.

When the device generates an interrupt:

- The current instruction finishes execution.
- The value of the program counter and status register are pushed onto the stack.
- The interrupt vector is translated into an address.
- The address is loaded into the program counter (PC) and the status register (SR) is altered to represent the interrupt level, masking out all interrupts of equal or lower priority.

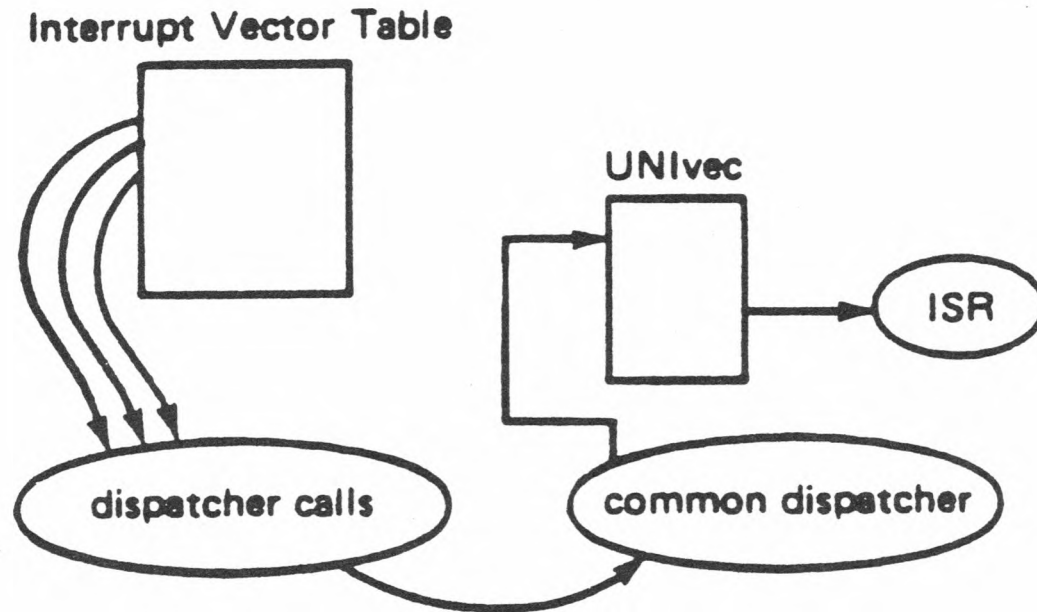


Figure 6. Interrupt Processing

Execution begins at the new address loaded in the PC, eventually resulting in the execution of a common dispatcher function which calls the ISR. All processing associated with the interrupt is executed by the interrupt service routine, called to service the interrupt.

The return from the ISR causes the dispatch function to restore the old value of SR and PC and execution continues in the old process.

INTERRUPT VECTORS AVAILABLE TO CUSTOMERS

A group of interrupt vectors are reserved exclusively for customer use.

Hex #	Decimal #	Hex Addr
be-ce	190-206	2f8-338

Figure 7. Customer Interrupt Vectors

Only controllers that can be strapped or programmed to generate these interrupt vectors can be integrated into the system. *Multibus* memory boards must respond to physical addresses in the range of 0x200000 - 0x3fffff (2 M). The task of mapping these boards into usable logical address space is described in *Chapter 10*.

Serial I/O Board

Chapter 7.

High Performance Serial I/O Board

OVERVIEW

This chapter provides a hardware description of the High Performance Serial I/O (HPSIO) Controller. The purpose of this description is to present the type of hardware and software information that must be available to a device driver writer. Additionally, Appendix A provides a portion of the functional specification for the HPSIO. Chapter 8 includes an example device driver that controls the HPSIO.

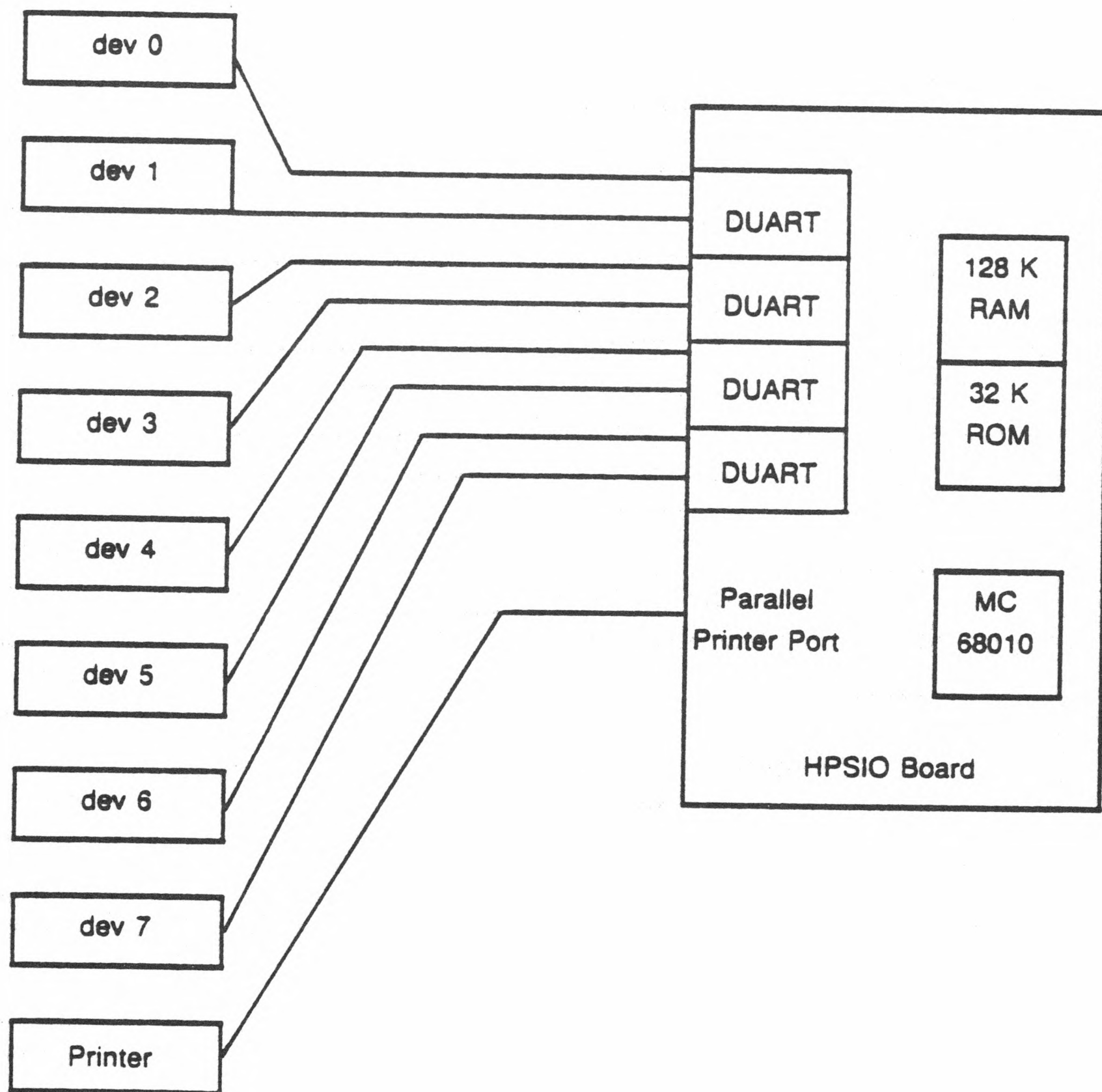


Figure 1. HPSIO Board

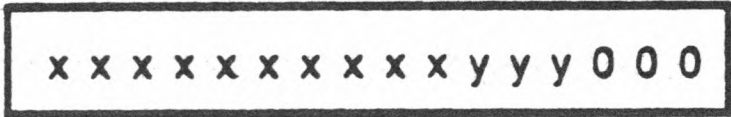
HPSIO FEATURES

The HPSIO board (Figure 1) contains:

- Four Dual Universal Asynchronous Receiver Transmitters (DUARTs) for the control of eight tty-type devices.
- One parallel printer port.
- An MC68010 processor.
- 32K of on-board ROM containing programs that can be performed by request from the PMC.
- 128K of on-board RAM that permits the downloading and executing of user programs.

HPSIO INTERFACE

The HPSIO board supports up to eight RS-232-C devices, such as terminals and line printers. The HPSIO controller also supports one parallel line printer. Communication between the HPSIO and the PMC is performed using 8 bytes of registers (Figure 3). The 8 bytes of registers are mapped through the PMC memory management unit (MMU) to *Multibus* I/O space. The base addresses of these registers are determined by the contents of a 16-bit register on the HPSIO board. This 16-bit register has the format shown in Figure 2.



- x Value determined by HPSIO board firmware (current value is 0x1400)
- y Value determined by three hardware straps.
- 0 Value of bit is zero (0).

Figure 2. HPSIO address register

To communicate with the HPSIO board using *Multibus* I/O:

1. Initialize the HPSIO interrupt vector register.
2. Initialize an I/O Parameter Block (IOPB) in memory.
3. For each HPSIO request (such as output), use an arbitration algorithm to access the HPSIO mailbox register. The IOPB address must be written to the mailbox register, and the mailbox register must be released.
4. If a response is expected from the HPSIO, wait for the requested function to complete.
5. If step 4 was required, decode the IOPB returned.

Register Definition		
Base Offset	Width	Definition
0	4	Mailbox Register
0	2	Mailbox Diagnostic Status
4	2	Interrupt Vector Register
6	2	Software Arbitration/Interrupt Status Register

Figure 3. HPSIO Registers

The HPSIO register set is described using the following C code:

```
struct chdev
{
    struct chIOPB      *mailbox;
    unsigned short     intrvector;
    unsigned short     arb;
} hp = (struct chdev *) 0x1f1400;
```

NOTE: The HPSIO board can access local system memory (LSM) directly to perform data transfer.

MAILBOX REGISTER

The mailbox register controls communication between the HPSIO board and the PMC board. The PMC requests a function to be performed by writing the address of an I/O Parameter Block (IOPB) into the mailbox register. An IOPB is a buffer in memory that describes the operation to be performed. All IOPBs have the same structure and are described in the IOPB section of this chapter.

The HPSIO performs the function described in an IOPB and then writes the IOPB address into the mailbox register. Each time the HPSIO writes to the mailbox register, the PMC is interrupted. Also, each time the PMC writes to the mailbox register, the HPSIO is interrupted.

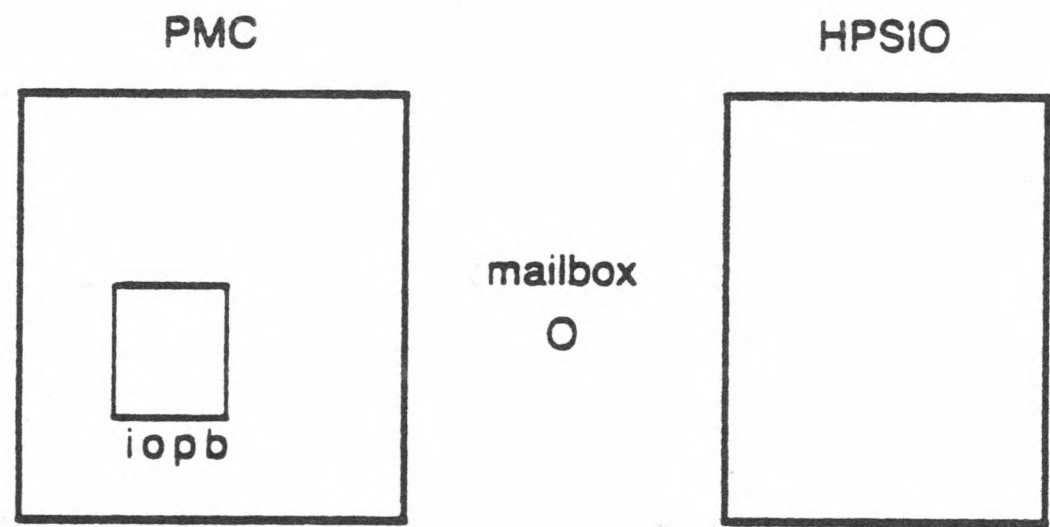


Figure 4. HPSIO/PMC communication

The following C code is performed to access the mailbox register for reading or writing:

```
hp->mailbox = IOPBp; /* write address of IOPB to mailbox */  
  
IOPBp = hp->mailbox; /* read address of IOPB from mailbox*/
```

INTERRUPT VECTOR REGISTER

The interrupt vector register must be initialized by the driver before an iobp is sent to the mailbox register. This is typically a start-of-day function.

The interrupt vector register values define how bus vectored interrupts are generated and contains the parameters shown in Figure 5.

Bits	Definition
0..7	Interrupt vector supplied during interrupt
8..a	Interrupt level (valid values are 0 through 6)
b	0 = Enable interrupts; 1 = Disable interrupts

Figure 5. Interrupt vector register

The following C code initializes the interrupt vector register so that the HPSIO generates vector number 0x70 at interrupt level 2 each time the HPSIO writes to the mailbox register.

```
hp->intrvector = 2<<8 + 0x70;
```

INTERRUPT STATUS/ARBITRATION REGISTER

The Interrupt Status/Arbitration Register contains 4-bits used to flag the existence of an interrupt, such as when the HPSIO has written to the mailbox register. Also, there are flags used to implement an arbitration algorithm.

Bits	Definition
0	If set, HPSIO has pending interrupt from PMC write to mailbox register
1	If set, PMC has pending interrupt from HPSIO write to mailbox register
6	ACK1 bit - if set, mailbox register in use
7	REQ2 bit for arbitration

Figure 6. Interrupt Status/Arbitration Register

NOTE: The PMC must adhere to the arbitration algorithm.

The arbitration algorithm is intended to prevent the PMC and HPSIO from using the mailbox register at the same time. The arbitration algorithm used by the PMC to access the mailbox register is:

- Turn on REQ2.
- Delay for a few cycles.
- Loop until ACK1 becomes 0.
- Access the mailbox register.
- Turn off REQ2.

The algorithm described above can be performed using this C code:

```
hp->arb = 0x80;                /* turn on ACK1 */
for (i=delay; i; --i)          /* delay a few cycles */
    ;
while (hp->arb & 0x40)          /* loop until ACK1 becomes 0 */
    ;
hp->mailbox = IOPBp;            /* access mailbox */ use the IOPB
hp->arb &= 0x80;                /* turn off REQ2 */
```

I/O PARAMETER BLOCK

An I/O Parameter Block (IOPB) is defined by the following C structure:

```
/*-----
 * hpsio firmware command io parameter buffer - one for each channel
 *-----
 */
struct chIOPB
{
    unsigned char    func;        /* function code */
    unsigned char    stat;        /* return status */
    unsigned char    chan;        /* channel number */
    unsigned char    cbaud;       /* trans. rate, signal control */
    unsigned char    cmode1;      /* parity type, char length */
    unsigned char    cmode2;      /* # stop bits, oper. mode */
    unsigned char    cstat;       /* line status */
    unsigned char    icntrl;      /* Interrupt Control/SRQ */
    unsigned char    dstat;       /* character status */
}
```



```
unsigned char    dchar;           /* received character */
char*    bufad;    /* buffer start address */
unsigned short  buflen;          /* buffer length */
};
```

NOTE:

One IOPB is dedicated to each HPSIO channel.

IOPB INTERFACE

The HPSIO firmware uses two types of data structures: LIOPB (load I/O parameter block) and CIOPB (channel I/O parameter block). Both the LIOPB and CIOPB are used to transmit commands to and receive commands from the system software. Both structures are memory resident and can be accessed using a pointer that is passed to the HPSIO firmware through the mailbox register.

The LIOPB is used for non-I/O specific functions, such as controller initialization, downloading, and return diagnostic log. Refer to Appendix A for further information on the LIOPB structure.

The CIOPB is used for I/O specific functions, such as:

- Channel Initialization
- Character Acknowledgement
- Configure Interrupt
- Output
- Line Change Acknowledgement

NOTE: The CIOPB is the IOPB referred to in the remaining sections of this book.

Writing the IOPB address to the mailbox register causes an interrupt to the HPSIO board firmware. The firmware reads the mailbox register contents, which clears the interrupt, and performs the function of the command contained in the IOPB. When the function has been completed successfully, the firmware updates the IOPB and writes the IOPB address to the mailbox register, if required. This write generates an interrupt to the PMC board, if HPSIO interrupts are enabled.

NOTE: The firmware does NOT overwrite the mailbox register contents.

Before any write is performed to the mailbox register, the firmware checks the Host Pending Interrupt bit in the Arbitration Register. If there is a pending interrupt, the firmware loops indefinitely until that interrupt is serviced. For example, until the mailbox register is read.

I/O COMPLETION - INCOMING IOPB'S

All IOPB requests coming in to the HPSIO board generate an interrupt. There are three types of IOPB requests:

- Type 1 requests are all HPSIO function requests that are returned with completion status information. All functions incoming to the HPSIO, except controller initialization and character acknowledge, are Type 1 functions. After the completion status information is placed in the IOPB for Type 1 functions, the IOPB address is written to the mailbox register.
- Type 2 requests are any unsolicited service requests from the HPSIO to the PMC, such as line change. The IOPB used to initialize an HPSIO channel is also used for a Type 2 function.

Consequently, the HPSIO firmware requires that an IOPB, used for channel initialization, be maintained as an unsolicited IOPB until the next initialization function for that channel.

REMEMBER: One IOPB is dedicated for each channel to handle all function requests for that channel.

Refer to Appendix A for a list of all HPSIO functions.

HPSIO COMMUNICATION CONCEPTS

There are two types of communication between the PMC and the HPSIO boards:

- Communication started by PMC
- Communication started by HPSIO

The same data flow is performed for each type of communication:

- A request is sent.
- A response is returned.

Function requests are always sent by the PMC.

Operation complete, in the status code of the IOPB, is the HPSIO positive response to a function request.

Service request, in the status code of the IOPB, is an unsolicited request from the HPSIO. When a terminal connected to the HPSIO sends a character, the HPSIO sends a service request to the PMC.

CHANNEL INITIALIZATION

Each channel on the HPSIO board must be initialized before any characters can be read from or written to

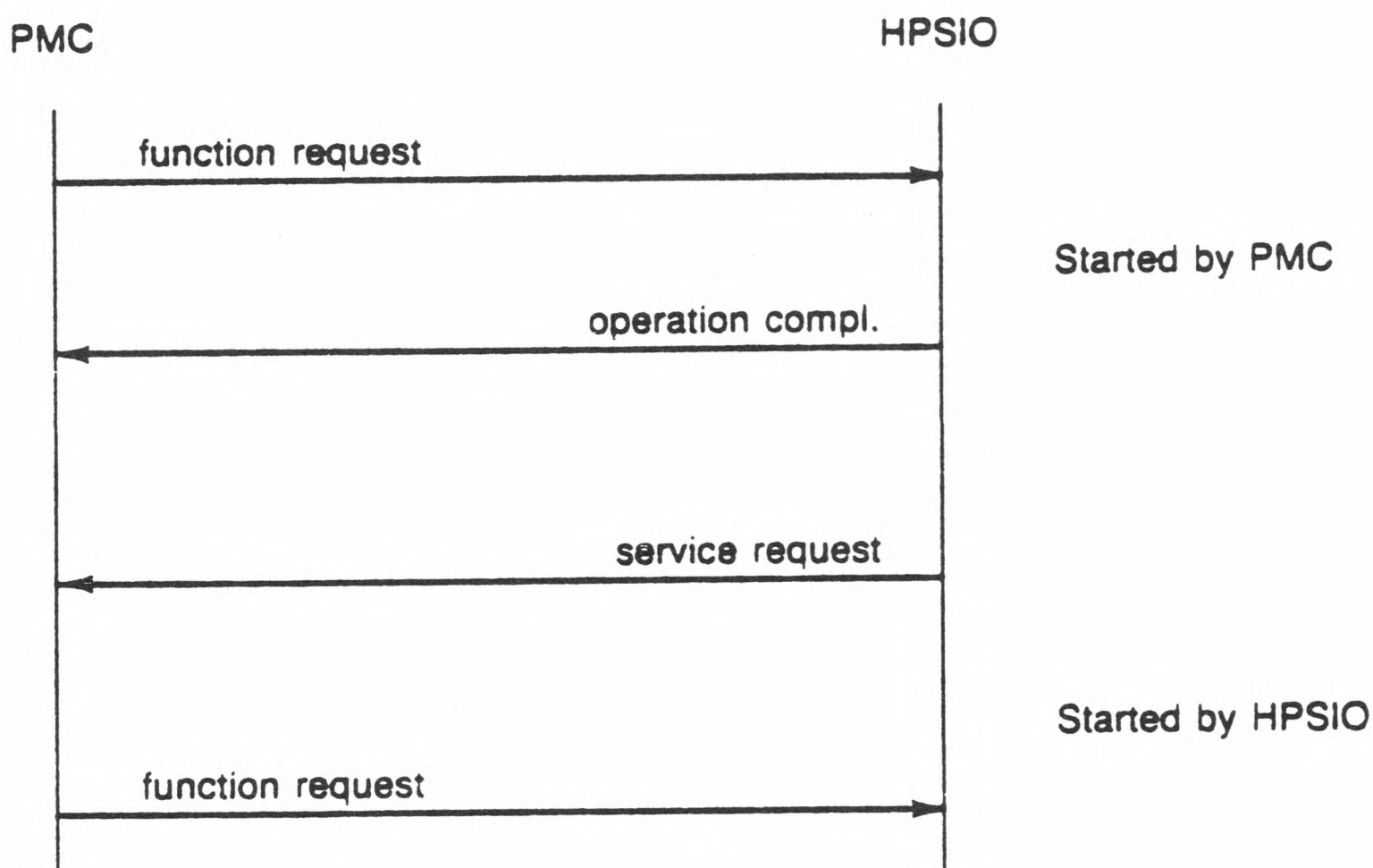


Figure 7. PMC/HPSIO protocol

that channel. Channel initialization is performed by the PMC by issuing an IOPB with the channel initialization function code. An IOPB address is written to the HPSIO mailbox register by the HPSIO firmware. Verification of the IOPB *stat* (status) field should reveal an "operation complete" status.

The following fields must be initialized in the IOPB before the channel initialization command can be performed.

unsigned char	func;	/* function code */
unsigned char	chan;	/* channel number */
unsigned char	cbaud;	/* transmission rate, signal control */
unsigned char	cmode1;	/* parity type, character length */
unsigned char	cmode2;	/* # stop bits, oper. mode */
unsigned char	icntrl;	/* Interrupt Control/SRQ */

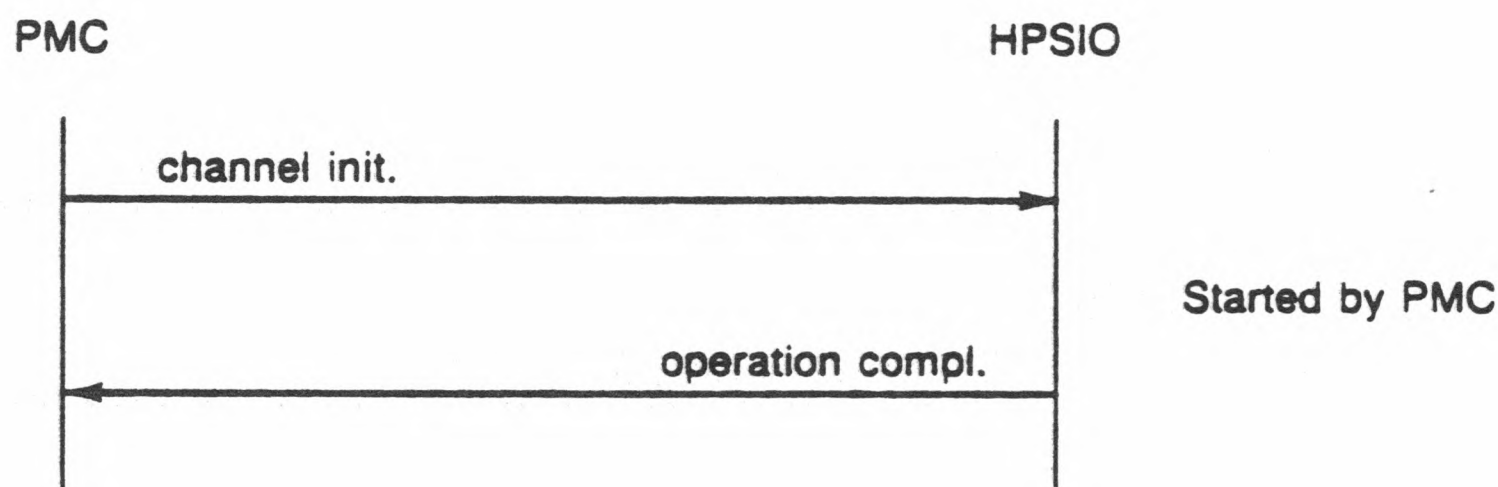


Figure 8. Channel initialization

Here's a brief description of each field:

func

Channel initialization (value 1).

chan

The channel number (0 through 7) for each serial port on the HPSIO.

cbaud

Baud rate, break on/off, input on/off, RTS on/off, DTR on/off.

cmodel

Character length and parity.

cmode2

Stop bit length, loopback, echo, non-echo operational mode.

icntrl

Receiver interrupt enable, input line change interrupt enable.

During channel initialization, the PMC performs these functions:

- Places the field values in the IOPB.
- Arbitrates for the mailbox register.
- Writes the IOPB address to the mailbox register.
- Frees the mailbox register.
- Waits for the HPSIO board to send a response.
- Validates the *stat* (status) field of the HPSIO response.

The HPSIO performs these functions:

- Initializes the channel
- Changes the *stat* (status) field of the IOPB to show success; 0 (operation complete) is the expected response.
- Writes the address of the IOPB into the mailbox register.

The IOPB used for channel initialization is also used for any unsolicited input requests.

OUTPUT

After a channel is initialized successfully, that channel can perform output. Typically, the PMC issues an IOPB with the output function code and expects the IOPB address to be written to the HPSIO mailbox register by the HPSIO firmware. Verification of the IOPB *stat* (status) field should reveal an "operation complete" status.

The following fields must be initialized in an IOPB for output:

unsigned char	func;	/* function code */
unsigned char	chan;	/* channel number */
char*	bufad;	/* buffer start address */
unsigned short	buflen;	/* buffer length */

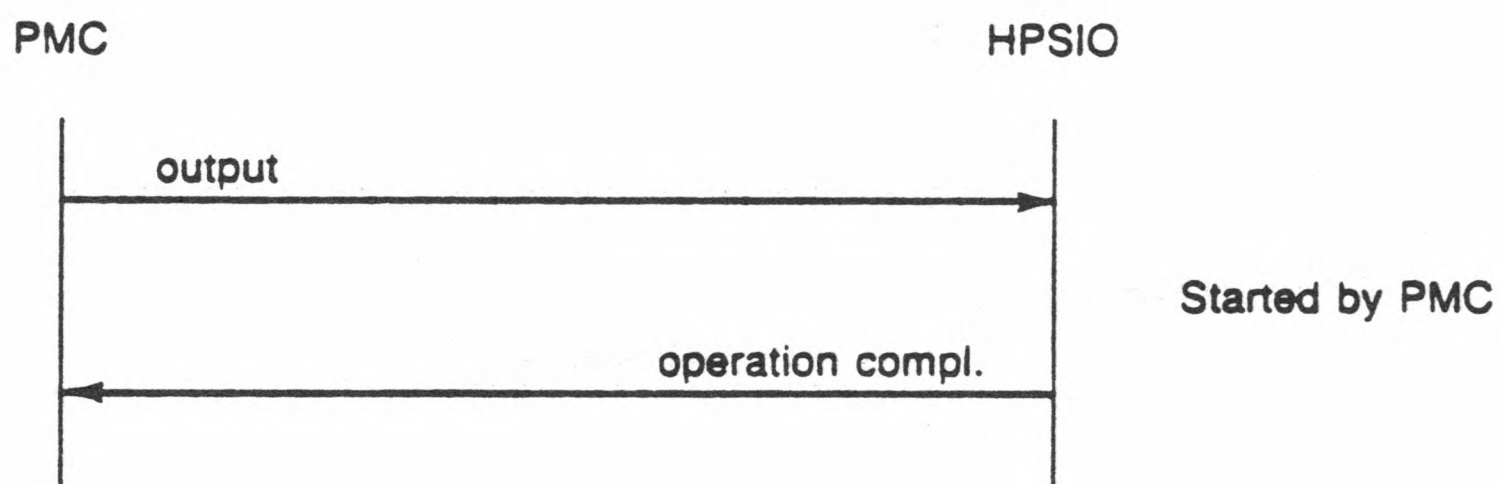


Figure 9. Output flow

Here is a brief description of each field:

func

Output (value 4).

chan

The channel number (0 through 7) for each serial port on the HPSIO.

bufad

Address of buffer to transmit.

buflen

Buffer length.

NOTE: The protocol is the same as for channel initialization.

INPUT

After a channel is initialized successfully, that channel can perform input.

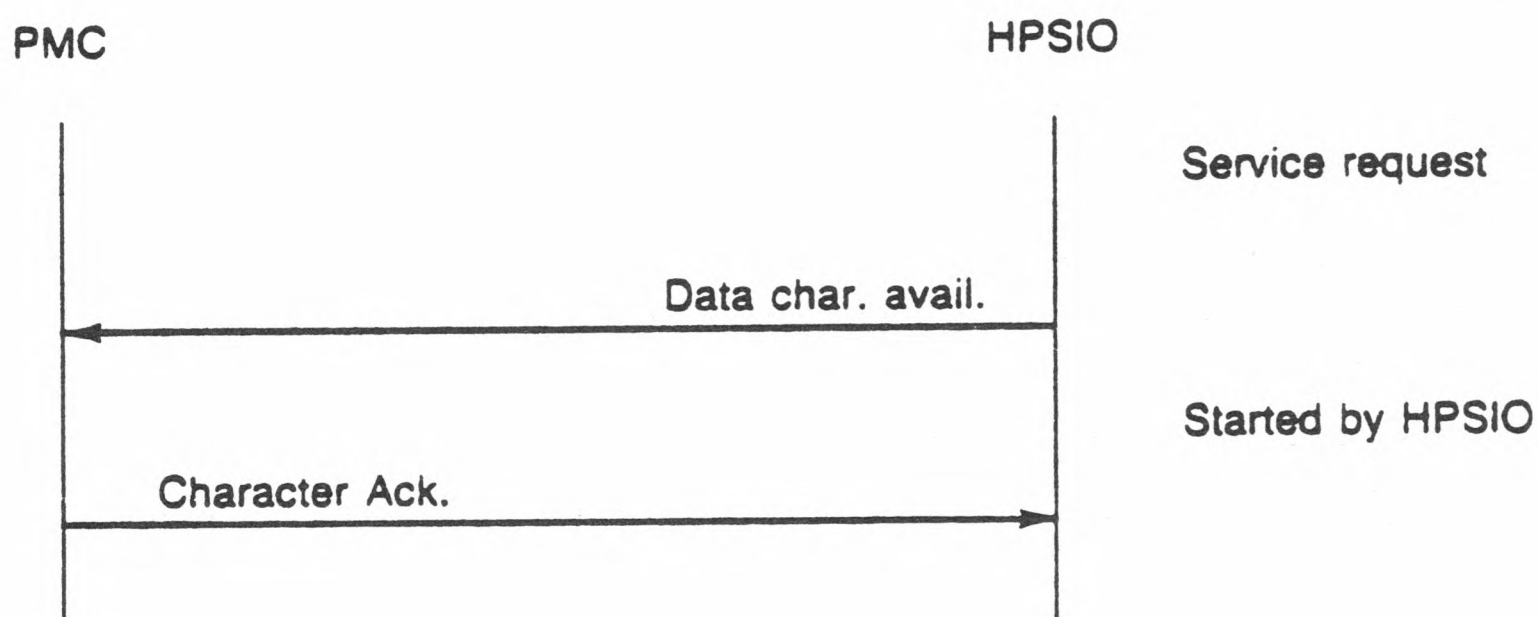


Figure 10. Input flow

The IOPB used during channel initialization is also used for all data character available service requests. The IOPB conditions include:

- Service request (value 12) is found in the IOPB *stat* (status) field.
- The Receiver Service Request bit in the *icntrl* field should be on.
- The character's status is found in the *dstat* field.
- The character is found in the *dchar* field.

The PMC must respond with a character acknowledgement command before another service request for the channel is mailed by the HPSIO.

CHARACTER ACKNOWLEDGEMENT

The character acknowledgement command informs the HPSIO firmware that the character has been read. If

another character is available in the HPSIO buffer in RAM, the firmware places the next character, along with its status, in the IOPB and puts a service request value in the *stat* (status) field.

After a character is received by the HPSIO from a terminal, the count of outstanding character acknowledgements is checked. If there are no acknowledgements outstanding, the IOPB is mailed to the PMC. If there is one or more outstanding acknowledgements, the new incoming character and its status is placed in the HPSIO's buffer in RAM for later retrieval.

When the last character acknowledgement IOPB is received (RAM buffer is empty), the count of outstanding acknowledgements is zero (0) and the IOPB is not returned.

The service request IOPB can be mailed at any point in the input flow with an unexpected error status resulting from a memory parity error or a bus error. These errors conditions are reported in the *stat* field.

The following fields must be initialized in an IOPB for a character acknowledgement.

```
unsigned char    func;           /* function code */
unsigned char    chan;          /* channel number */
```

Here is a brief description of each field:

func

Character acknowledgement (value 2).

chan

The channel number (0 through 7) for each serial port on the HPSIO.

Chapter 8.

A Character Device Driver

OVERVIEW

This chapter describes a device driver that controls the HPSIO board. Processes can read and write buffers of characters to tty devices attached to the HPSIO using this driver. Following a general overview, each driver routine is explained.

DRIVER OVERVIEW

Figure 1 shows the relationship between the various components of a character device driver. The dashed lines show control flow. Both function calls and wakeups control the execution of the driver code. The continuous lines show data flow to and from the controller.

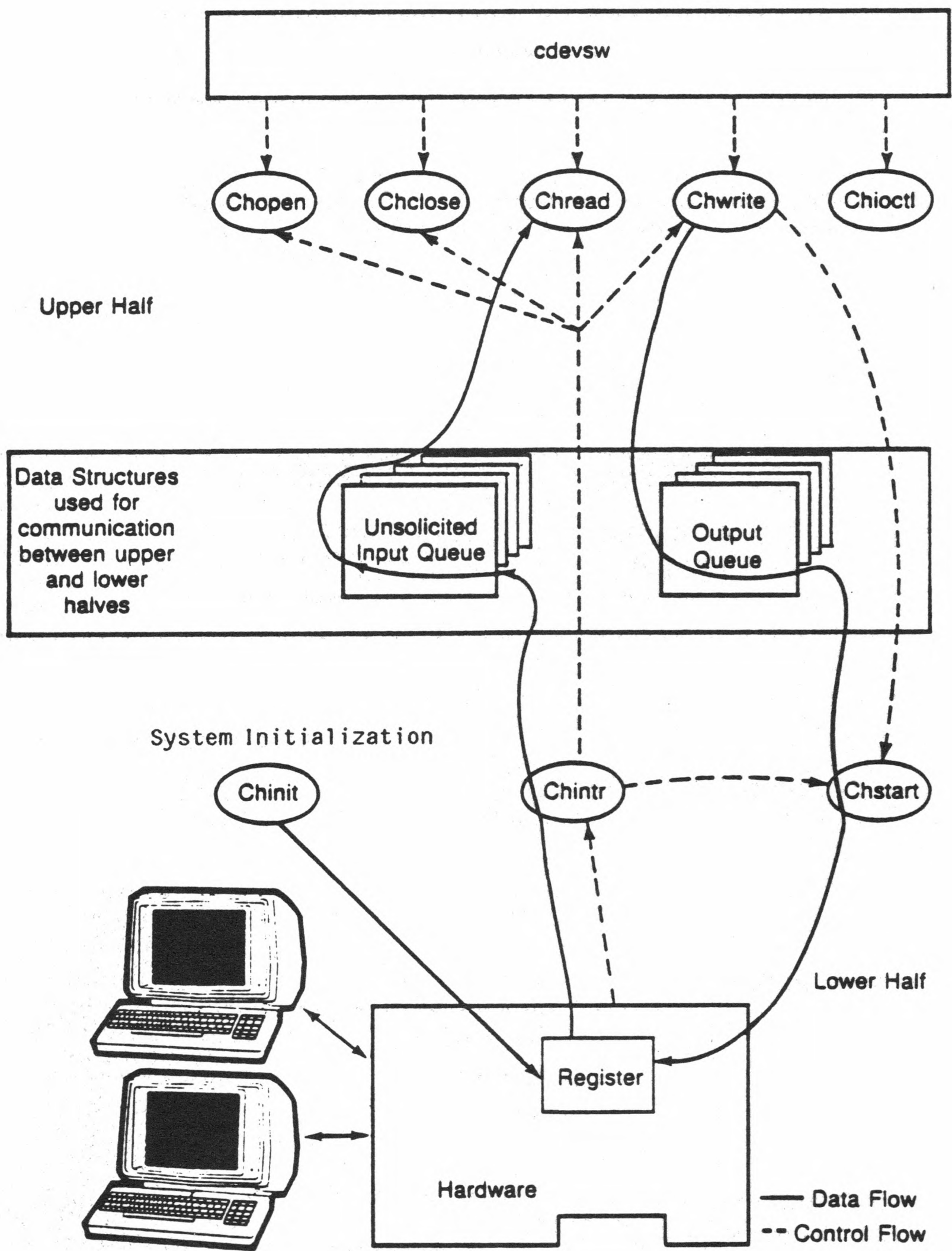


Figure 1. Driver Block Description

Control Flow Example

A process identifies the *chread* routine through the *cdevsw* table. The process executing *chread* may be put to sleep if more characters are requested than are currently available in the unsolicited input queue. The interrupt service routine, *chintr*, is invoked when a terminal user presses a key. When *chintr* has put enough characters into the unsolicited input queue to satisfy the process's request, it wakes up the process.

Data Flow Example

A process invokes the *chwrite* routine to send a buffer of information to a terminal. This buffer is copied to the output queue. *Chstart* takes characters off the output queue and sends them to the controller.

Lower Half Output Logic

The output portion of the lower half of the driver is divided into two routines: the *chstart* routine writes to the adapter's registers to do the physical output; after the adapter generates an output complete interrupt, the ISR routine--*chintr*--calls the *chstart* routine to begin the next output. When the output queue becomes empty, the lower half is temporarily finished with output and is considered to be not busy. When *chwrite* moves characters to the output queue and the lower half is not busy, it must call *chstart* to start output.

Chopen Definition

The open routine, *chopen*, is called using the *cdevsw* table. This routine is separated into four sections:

1. Initial requirements
2. Protect
3. Channel initialization
4. Unprotect

Initial Requirements Section

The initial requirements section permits or doesn't permit the open. If all the requirements are satisfied, the open is permitted. If all the requirements aren't met, the value of *u.u_error* is set and the routine stops.

Protect Section

The protect section protects the channel initialization section. The HPSIO can not have two channel initialization functions in progress at the same time. For example, if a channel initialization function is sent to channel 0, a channel initialization function can not be sent to channel 1 until the channel initialization acknowledgement is received for channel 0.

If the protect section detects a channel initialization in progress (indicated by the flag `COPNPRG`), it puts the process to sleep. If a process is put to sleep in the protect section, there must be a process executing code in the channel initialization section. As the process that is in the channel initialization section passes through the unprotect section, it issues a wakeup to processes put to sleep in the protect section.

Channel Initialization Section

The channel initialization section constructs a channel initialization IOPB. The address of the IOPB is mailed to the HPSIO. The HPSIO initializes the channel and sends back a channel initialization acknowledgement. The process waiting for a channel initialization acknowledgement is put to sleep. The ISR routine, *chintr*, calls wakeup for the process when the channel initialization acknowledgement for this channel is received.

Unprotect Section

The unprotect section wakes up processes put to sleep in the protect section. When a process reaches this section, it has finished with the channel initialization and another channel can be initialized. All processes put to sleep in the protect section are issued a wakeup by processes passing through the unprotect section.

CHOPEN LISTING

```
chopen(minor_d, flag)
    dev_t minor_d;    /* minor device */
    int flag;          /* open flag, bit 0 read, bit 1 write */
{
    struct cinfo *cp;    /* cinfo pointer for channel */
```


Chapter 8

```
int priority;          /* incoming priority */
struct ciopb *iopb;    /* iopb used locally */

printf("open %x.", minor_d);
/*----- initial requirements -----*/
if (minor_d >= ch_cnt) /* validate the minor device number */
    u.u_error = EIO;
switch (flag & 0x03) /* read/write flag */
{
    case (0):          /* no read / no write */
    case (3):          /* read / write */
        u.u_error = EINVAL;
        break;
    case (1):          /* read / no write */
        if (cstate & COPENWRITE)
            u.u_error = EINVAL;
        else
            cstate |= COPENREAD;
        break;
    case (2):          /* no read / write */
        if (cstate & COPENREAD)
            u.u_error = EINVAL;
        else
            cstate |= COPENWRITE;
        break;
}

cp = &ch_cinfo[minor_d]; /* make sure it is not already open */
if (cp->state & COPEN)
    u.u_error = EBUSY;

if (u.u_error)
    return;

cp->state = COPEN; /* this channel is exclusively open */

/*----- protect -----*/
priority = spl_c();
while(cstate & COPNPRG)
{
    /* while other open is in progress */
    printf("sleeping in protect %x\n", minor_d);
    cstate |= COPNSLP; /* mark as sleeping */
    sleep (&cstate, CPRI);
    printf("wakeup in protect %x\n", minor_d);
}
cstate |= COPNPRG; /* mark open in progress */

/*----- channel initialization -----*/
iopb = &ch_ciopb_io[minor_d].input;
iopb->func = init_iopb.func;
```

```

iopb->chan = minor_d;
iopb->cbaud = init_iopb.cbaud;
iopb->cmode1 = init_iopb.cmode1;
iopb->cmode2 = init_iopb.cmode2;
iopb->icntrl = init_iopb.icntrl;

cp->state != CCIWAIT;      /* mark as waiting for chan init ack */
carb(hp);                  /* write the command to the HPSIO mailbox */
cmboxwrite(iopb, hp);
cmboxfree(hp);
while(cp->state & CCIWAIT) /* waiting for the ISR to receive a */
{                           /* channel init ack. */
    cp->state != CCISLP;    /* mark as sleeping for a channel init ack */
    sleep(cp, CPRI);
}

/*----- unprotect -----*/
cstate &= COPNPRG;          /* open is no longer in progress */
if (cstate & COPNSLP)       /* some other processes are waiting to open */
{
    cstate &= COPNSLP;      /* now they can be awake */
    wakeup (&cstate);
}
splx(priority);
printf("d");
}

```

CHREAD DEFINITION

The read routine, *chread*, is called using the *cdevsw* table. Characters are taken off the unsolicited input queue and moved to a buffer in the program portion of the process. The *clist*, *clin*, represents the unsolicited input queue. If there aren't enough characters in the unsolicited input queue, the process is put to sleep until enough characters are enqueued.

The number of characters to transfer is in *u.u_base* when the process is executing *chread*. The *u.u_base* value is copied to the variable *ccntin* in *chread*. The ISR, *chintr*, enqueues characters as they are entered and wakes up the process sleeping in *chread* when *clin* has enough characters to satisfy the request.

Chapter 8

The `u.u_base` value is copied to `ccntin` because a different `u` structure is active when the ISR is executing.

CHREAD LISTING

```
chread(minor_d)
dev_t minor_d;
/*-----
 * transfer characters from kernel clist to user space.
 * the user must request the number of characters desired.
 *
 * if not enough characters in the unsolicited input queue
 *     sleep
 * move characters to user space
 *
 *-----
 */
{
    struct cinfo * cp;           /* info structure */
    struct clist * cl;           /* pointer to clist */
    int priority;                /* previous priority */

    cp = &ch_cinfo[minor_d];
    cl = &cp->clin;              /* unsolicited input queue */
    cp->ccntin = u.u_count;      /* number of characters requested */

    priority = spl_c();          /* critical section */
    while (cp->ccntin > cl->c_cc)
    {
        printf("chread: sleeping for input\n");
        cp->state != CINSLP;     /* sleep until more chars are typed */
        sleep(cl, CPR1);
    }
    splx(priority);

    while (passc(getc(cl)) != -1) /* pass to user space the characters */
        ;                       /* in the clist */
}
```


CHWRITE DEFINITION

The write routine, *chwrite*, is called using the *cdevsw* table. Characters are copied from a program buffer into the output queue. The *clist--clout--*represents the output queue. A few characters are copied to the output queue, output is started if the minor device is not busy, and sleeps if the output queue reaches a high water mark. The steps are repeated until all of the buffer has been transferred to the output queue. The ISR issues a wakeup to a process sleeping in *chwrite* when the output queue goes below the low water mark.

CHWRITE LISTING

```
chwrite(minor_d)
dev_t minor_d;
/*-----
 * transfer characters from user space to kernel clist and start output.
 * while (characters to move)
 *   move cfew characters clist
 *   start output if not busy
 *   if more characters to move
 *     if number of characters in clist > CHIWATER
 *       sleep
 *
 *-----
 */
{
    struct cinfo * cp;           /* info structure */
    struct clist * cl;           /* pointer to clist */
    int priority;                /* previous priority */
    int c;                       /* character from user space */
    int count;                   /* temp counter */

    printf("wr %x\n", minor_d);
    cp = &ch_cinfo[minor_d];
    cl = &cp->clout;

    c = cpass();
    while (c != -1)               /* while characters to read */
    {                             /* move a few chars to clist */
        for(count=cfew; (c != -1) && count; count-- )
```

```
        {
            chputc(c, cl);          /* chputc is not a kernel call */
            c = cpass();            /* put the char in clist, maybe sleep */
            c = cpass();            /* get the next char */
        }

    if(!(cp->state & CBUSY)) /* start output */
    {
        chstart(minor_d);
    }

    if (c != -1) /* about to fall through */
        priority = spl_c();
    while (cl->c_cc > CHIWATER) /* wait for output to drain */
    {
        printf("ws %x.", minor_d);
        cp->state |= COSLP;
        sleep(cl, CPRI);
        printf("ww %x.", minor_d);
    }
    splx(priority);
}
}
```

CHCLOSE DEFINITION

The close routine, *chclose*, is called using the *cdevsw* table. The process executing *chclose* must be suspended until all characters in the output queue have been given to the controller. The output queue is empty if it has zero characters. The ISR issues a wakeup to processes sleeping in *chclose* when the output queue is empty.

CHCLOSE LISTING

```
chclose(minor_d)
{
    struct cinfo *cp;          /* pointer to cinfo structure */
    struct clist *cl;          /* pointer to clist */
    int priority;              /* processor priority level */

    cp = &ch_cinfo[minor_d];
    cl = &cp->clout;
```

```

priority = spl_c();
while (cl->c_cc > 0)          /* wait for output to drain completely */
{
    cp->state != CCLSSLP;
    sleep(cl, CPRI);
}
cp->state = 0;                /* reset for the next process to open */
splx(priority);
}

```

CHINIT DEFINITION

The initialization routine, *chint*, is executed once during boot up initialization. The interrupt vector register on the HPSIO is initialized.

CHINIT.C LISTING

```

/*-----
 * external references
 *-----
 */
extern int ch_cnt; /* number of channels */
extern struct cinfo ch_cinfo[]; /* information structure */
extern struct ciopb_io ch_ciopb_io[]; /* input/output iopb array */

struct cdev *hp = CREGBASE;

/* channel initialization command */
struct ciopb init_iopb =
/* func stat chan cbaud cmode1 cmode2 cstat icntrl dstat dchar bufad buflen */
{CFCHANINIT,0,0,0xeb, 0x02, 0x07, 0, 0x01, 0, 0, 0, 0};

int cfew = CFEW; /* number of characters to write at a time */
int cstate = 0; /* state of device as a whole see COPNSLP */

chinit()
/*-----
 * initialization routine.
 * the interrupt vector register needs to be initialized with
 * the interrupt vector, priority level (0..7), and enable interrupts
 *-----
 */
{
    hp->intrvector = CINTLEVEL<<8 | CVECTOR;
}

```



```
printf("c started interrupt vector %x\n", hp->intrvector);
}
```

CHSTART DEFINITION

The start routine, *chstart*, is the output portion of the lower half of the driver. A *cblock* is taken off the output *clist* and an output IOPB is constructed using the *cblock*. It is important that this *cblock* be returned to the freelist. An interrupt is generated when the HPSIO has completed the output operation, so the ISR must perform clean up functions.

It is important to understand the reason for the start routine. At first glance, it may seem that the start function should be contained within the ISR. Each time an output request is completed, it's time to begin another output. A problem arises when the device is idle. The write routine needs a method to "start" the output after adding to an empty queue. In short, the start routine is the code necessary to start an output operation; it is called by the ISR and the write routine.

CHSTART LISTING

```
chstart(minor_d)
dev_t minor_d;
/*-----
 * start io on one of the channels,
 * the channel is not busy - guaranteed
 * if there is a block on clist
 * write output iopb for list
 * mark state as busy
 *
 * note that the block taken off list is put in a the variable: cblock
 * so that the ISR may return it to the free list
 *-----
 */
{
    struct clist * cl;          /* clist */
```

```
struct cinfo * cp;           /* c information */
struct ciopb * iopb;         /* iopb filled for output */
struct cblock * bp;          /* temporary cblock pointer */

cp = &ch_cinfo[minor_d];
cl = &cp->clout;
iopb = &ch_ciopb_io[minor_d].output;
/* if there are no more cblocks */
if ((bp = cp->cbout = getcb(cl)) == 0)
    panic("chstart called with nothing to print\n");

iopb->func = CFOUTPUT; /* make iopb */
iopb->chan = minor_d;
iopb->bufad = &bp->c_data[bp->c_first];
iopb->buflen = bp->c_last - bp->c_first;

cp->state != CBUSY; /* mark as busy */

carb(hp); /* write to mailbox register */
cmboxwrite(iopb, hp);
cmboxfree(hp);
}
```

CHINTR DEFINITION

Chintr is the ISR that handles all interrupts. There are three types of IOPB's that can be made by the HPSIO and delivered to the mailbox:

- Channel initialization acknowledgement
- Character typed
- Output acknowledgement

There is an input and output IOPB for each channel. The input IOPB is first used for channel initialization and channel initialization exchange. After initialization, the input IOPB is used for the service request (for example, a character entered from the terminal) and character acknowledge exchange.

The output IOPB is used for the output request and output acknowledge exchange.

Channel Initialization Acknowledgement (CFCHANINIT)

When the channel initialization is completed, the HPSIO sends back a channel initialization acknowledgement. An interrupt is generated, *chintr* examines the IOPB returned, and it is verified. If a process is asleep in the *chopen* routine waiting for a channel initialization acknowledgement, the process is issued a wakeup. The function code is set to CFCHARACK because all remaining functions for this IOPB are for character acknowledgements.

Character Typed (CFCHARACK)

A character entered at the keyboard causes the HPSIO to send a service request. A service request is unsolicited input and, therefore, needs to be added to the unsolicited input queue. The IOPB is verified. If a process is asleep in the *chread* routine, it is issued a wakeup if enough characters are on the unsolicited input queue.

Output Acknowledgement (CFOUTPUT)

After the HPSIO has completed processing an output request, an output acknowledgement IOPB is delivered to the PMC. The *cblock* used for the output must be put back on the free list.

If a process is asleep in the driver's *chwrite* routine waiting for the output *clist* to become empty, the process is issued a wakeup when the queue is below the low water mark. If a process is put to sleep in the driver's *chclose* routine waiting for the output *clist* to become empty, the process is issued a wakeup when

the queue becomes empty. If there are more characters in the output *clist*, the *chstart* routine must be called to start the output of another *cblock*.

CHINTR LISTING

```
chintr()
/*-----
 * ISR
 * arbitrate for the mailbox
 * read the mailbox register
 * case(Acknowledged Channel Init CIOPB)
 *   validate the iopb
 *
 * case(Acknowledged output)
 *   validate the iopb
 *   return the cblock that was printed back to the free list
 *   wakeup upper half if needed
 *   call start to start the next cblock
 *
 * free the mailbox
 *
 *-----
 */
{
    struct ciopb * iopb;          /* iopb in the mailbox */
    struct cinfo * cp;            /* info structure that caused int */
    struct clist * cl;            /* clist pointer */
    int minor_d;                  /* minor device number */
    char tmp_func;                /* temporary function holder */
    int priority;                 /* priority level */

    carb(hp);                     /* ask for use of the mailbox */
    cmbxread(iopb, hp);           /* read the mailbox */
                                  /* iopb numer (ie channel number) */
    minor_d = (iopb - (struct ciopb *)ch_ciopb_io)/2;
    cp = &ch_cinfo[minor_d];      /* info structure pointer */
    printf("i%x", minor_d);

    switch (iopb->func)
    {
        /*-----*/
        case (CFCHANINIT):        /* Acknowledged Channel Init CIOPB */
            /*printf("ci");*/
            if (iopb->stat != CSOPCPL)
            {

```

```

        clogerr(iopb, "channel init failed: operation not complete");
        break;
    }

    /*-- channel initialization wakeup --*/
    cp->state &= CCIWAIT; /* no longer waiting for chan init ack */
    if(cp->state & CCISLP) /* if sleeping for a chan init ack */
    {
        cp->state &= CCISLP; /* no longer sleeping for chan init ack */
        wakeup(cp);         /* wakeup processes waiting for */
    }                       /* chan init ack */

    iopb->func = CFCHARACK; /* from now on it will be input */
    break;

/*-----*/
case (CFCHARACK):          /* character has been typed */
    if(! (iopb->icntrl & CRCVSR))
        pr_iopb(iopb, "char ack but no CRCVSR");
    printf("in char");
    cl = &cp->clin;         /* unsolicited input queue */
    putc(iopb->dchar, cl);  /* store the character away */

    cmboxfree(hp);
    carb(hp);              /* ask for use of the mailbox */
    priority = spl0();
    splx(priority);
    cmboxwrite(iopb, hp);  /* write a char ack to the hpsio */
    /*-- input wakeup --*/
    if (cp->state & CINSLP) /* process sleeping on input */
        if (cl->c_cc >= cp->ccntin) /* worth waking up */
        {
            cp->state &= CINSLP;
            wakeup(cl);
        }
    break;

/*-----*/
case (CFOUTPUT):          /* acknowledged output */
    printf("o%x", minor_d);
    if (iopb->stat != CSOPCPL)
    {
        clogerr(iopb, "output failed: operation not complete");
        break;
    }

    /*-- output wakeup --*/
    putcf(cp->cbout);      /* return the cblock to free list */
    if (cp->clout.c_cc <= CLOWATER) /* wake up top half if drained */
        if(cp->state & COSLP) /* and if sleeping for output */
        {
            cp->state &= COSLP;
        }

```

```
        wakeup(&cp->clout);
    }
    if (cp->clout.c_cc == 0) /*-- close wakeup --*/
    {                         /* wake close if drained to zero */
        cp->state &= CBUSY;   /* no longer busy */
        if(cp->state & CCLSSLP) /* and if sleeping for close */
        {
            cp->state &= CCLSSLP;
            wakeup(&cp->clout);
        }
    }
    else
        chstart(minor_d);    /* start up the next transaction */
    break;

default:
    clogerr(iopb, "unexpected interrupt");
    break;
} /* end switch */

cmboxfree(hp);
printf("e");
}
```

MASTER/DFILE ENTRIES

The following lines in the *master* file describe the *ch* driver.

*1	2	3	4	5	6	7mb	8mc	9	10	11	12
ch	4	137	4	ch	8	0	21	8	4	cinfo	ciopb_io

This line in the *dfile* for the *config* program describes the *ch* driver:

ch	700	761200
----	-----	--------

CREATING/TESTING THE KERNEL

The following commands create the device nodes:

```
# /etc/mknod /dev/ch0 21 0
# /etc/mknod /dev/ch1 21 1
# /etc/mknod /dev/ch2 21 2
# /etc/mknod /dev/ch3 21 3
# /etc/mknod /dev/ch4 21 4
# /etc/mknod /dev/ch5 21 5
# /etc/mknod /dev/ch6 21 6
# /etc/mknod /dev/ch7 21 7
```

This makefile creates the kernel:

```
test    =    ch.o

#   define for name.o so boot up console messages are printed out correctly
SYS      =    UTS V
VER =    yours
NODE     =    S5000
REL =    1R1
#   MACH = 68010 for 5000/20 and 5000/40
#   MACH = 68020 for 5000/20 and 32-bit 5000/40
MACH     =    68010
DEFS     =    -DSYS="\$(SYS)\"" \
              -DVER="\$(VER)\"" \
              -DNODE="\$(NODE)\"" \
              -DREL="\$(REL)\"" \
              -DMACH="\$(MACH)\""

AS       =    as
CC       =    cc
#   RELOC = 108000 for 5000/20 and 5000/40
#   RELOC = E00000 for 5000/50 and 32-bit 5000/40
RELOC    =    108000**
CFLAGS   =    -K -I. -I$(INCRT)
LFLAGS   =    -R $(RELOC) -N -e susentry

LIBS     =    /kernel/sperry
INCRT    =    /usr/include
MASTER  =    /usr/acct/yours/master

.c.o     $(CC) $(CFLAGS) -c $<

.s.o     $(AS) $(ASFLAG) -o $*.o $*.s
```

```
tape:  unix
      bootmedia t unix
```

```
flex:  unix
      bootmedia f unix
```

```
unix:  conf.o name.o univec.o $(LIBS)/lib[0-9] $(test)
      ld $(LFLAGS) /kernel/sperry/ml/start.o\
          $(test)\
```

```
      \
      univec.o\
      conf.o\
      name.o\
      /usr/acct/yours/linesw.o\
      $(LIBS)/lib[0-9]\
      $(LIBS)/lib3\
      -o unix\
      2> undefs
```

```
name.o: $(INCR)/sys/utsname.h
      $(CC) $(CFLAGS) $(DEFS) -c name.c
```

```
conf.o: conf.c
```

```
univec.o: univec.c
```

```
conf.c: config.cf $(MASTER)
      config -m $(MASTER) config.cf
      fixconf
```

```
ch.o:  ch.h\
      ch.c\
      chclose.c\
      chinit.c\
      chintr.c\
      chioctl.c\
      chopen.c\
      chread.c\
      chrest.c\
      chstart.c\
      chwrite.c
```

Chapter 8

This program displays a file on terminal 2:

```
$ cp /etc/termcap /dev/ch2
```

COMPLETE SOURCE LISTING

```
/*
 * ch.c -- A character device driver
 */

#include "sys/param.h"
#include "sys/types.h"
#include "sys/dir.h"
#include "sys/signal.h"
#include "sys/user.h"
#include "sys/errno.h"
#include "sys/tty.h"

#include "ch.h"

#include "chinit.c"
#include "chopen.c"
#include "chwrite.c"
#include "chread.c"
#include "chstart.c"
#include "chintr.c"
#include "chclose.c"
#include "chioc1.c"
#include "chrest.c"
/*-----
 * external references
 *-----
 */
extern int ch_cnt; /* number of channels */
extern struct cinfo ch_cinfo[]; /* information structure */
extern struct ciopb_io ch_ciopb_io[]; /* input/output iopb array */

struct cdev *hp = CREGBASE;

/* channel initialization command */
struct ciopb init_iopb =
/* func stat chan cbaud cmode1 cmode2 cstat icntrl dstat dchar bufad buflen */
{CFCHANINIT,0,0, 0xeb, 0x02, 0x07, 0, 0x01, 0, 0, 0, 0};

int cfew = CFEW; /* number of characters to write at a time */
int cstate = 0; /* state of device as a whole see COPNSLP */
```



```

chinit()
/*-----
 * initialization routine.
 * the interrupt vector register needs to be initialized with
 * the interrupt vector, priority level (0..7), and enable interrupts
 *-----
 */
{
    hp->intrvector = CINTLEVEL<<8 | CVECTOR;
    printf("c started interrupt vector %x\n", hp->intrvector);
}

chopen(minor_d, flag)
    dev_t minor_d;    /* minor device */
    int flag;          /* open flag, bit 0 read, bit 1 write */
{
    struct cinfo *cp;    /* cinfo pointer for channel */
    int priority;        /* incoming priority */
    struct ciopb *iopb;  /* iopb used locally */

    printf("open %x.", minor_d);
    /*----- initial requirements -----*/
    if (minor_d >= ch_cnt)    /* validate the minor device number */
        u.u_error = EIO;
    switch (flag & 0x03)      /* read/write flag */
    {
        case (0):            /* no read / no write */
        case (3):            /* read / write */
            u.u_error = EINVAL;
            break;
        case (1):            /* read / no write */
            if (cstate & COPENWRITE)
                u.u_error = EINVAL;
            else
                cstate |= COPENREAD;
            break;
        case (2):            /* no read / write */
            if (cstate & COPENREAD)
                u.u_error = EINVAL;
            else
                cstate |= COPENWRITE;
            break;
    }

    cp = &ch_cinfo[minor_d]; /* make sure it is not already open */
    if (cp->state & COPEN)
        u.u_error = EBUSY;

    if (u.u_error)
        return;

```

Chapter 8

```
cp->state = COPEN;          /* this channel is exclusively open */

/*----- protect -----*/
priority = spl_c();
while(cstate & COPNPRG)
{
    /* while other open is in progress */
    printf("sleeping in protect %x\n", minor_d);
    cstate != COPNSLP;      /* mark as sleeping */
    sleep (&cstate, CPRI);
    printf("wakeup in protect %x\n", minor_d);
}
cstate != COPNPRG;          /* mark open in progress */

/*----- channel initialization -----*/
iopb = &ch_ciopb_io[minor_d].input;
iopb->func = init_iopb.func;
iopb->chan = minor_d;
iopb->cbaud = init_iopb.cbaud;
iopb->cmode1 = init_iopb.cmode1;
iopb->cmode2 = init_iopb.cmode2;
iopb->icntrl = init_iopb.icntrl;

cp->state != CCIWAIT;       /* mark as waiting for chan init ack */
carb(hp);                  /* write the command to the HPSIO mailbox */
cmboxwrite(iopb, hp);
cmboxfree(hp);
while(cp->state & CCIWAIT) /* waiting for the ISR to receive a */
{
    /* channel init ack. */
    cp->state != CCISLP;    /* mark as sleeping for a channel init ack */
    sleep(cp, CPRI);
}

/*----- unprotect -----*/
cstate &= COPNPRG;          /* open is no longer in progress */
if (cstate & COPNSLP)       /* some other processes are waiting to open */
{
    cstate &= COPNSLP;      /* now they can be awake */
    wakeup (&cstate);
}
splx(priority);
printf("d");
}
chwrite(minor_d)
dev_t minor_d;

/*-----
* transfer characters from user space to kernel clist and start output.
* while (characters to move)
*   move cfew characters clist
*   start output if not busy
*   if more characters to move
*/
```

```

*      if number of characters in clist > CHIWATER
*      sleep
*
*-----
*/
{
    struct cinfo * cp;          /* info structure */
    struct clist * cl;          /* pointer to clist */
    int priority;               /* previous priority */
    int c;                      /* character from user space */
    int count;                  /* temp counter */

    printf("wr %x\n", minor_d);
    cp = &ch_cinfo[minor_d];
    cl = &cp->clout;

    c = cpass();
    while (c != -1)              /* while characters to read */
    {                             /* move a few chars to clist */
        for(count=cfew; (c != -1) && count; count-- )
        {
            chputc(c, cl);      /* chputc is not a kernel call */
            /* put the char in clist, maybe sleep */
            c = cpass();        /* get the next char */
        }

        if(!(cp->state & CBUSY)) /* start output */
        {
            chstart(minor_d);
        }

        if (c != -1)             /* about to fall through */
            priority = spl_c();
        while (cl->c_cc > CHIWATER) /* wait for output to drain */
        {
            printf("ws %x.", minor_d);
            cp->state |= COSLP;
            sleep(cl, CPRI);
            printf("ww %x.", minor_d);
        }
        splx(priority);
    }
}

chread(minor_d)
dev_t minor_d;
/*-----
* transfer characters from kernel clist to user space.
* the user must request the number of characters desired.
*
* if not enough characters in the unsolicited input queue
* sleep

```


Chapter 8

```
* move characters to user space
*
*-----
*/
{
    struct cinfo * cp;           /* info structure */
    struct clist * cl;           /* pointer to clist */
    int priority;                /* previous priority */

    cp = &ch_cinfo[minor_d];
    cl = &cp->clin;               /* unsolicited input queue */
    cp->ccntin = u.u_count;       /* number of characters requested */

    priority = spl_c();           /* critical section */
    while (cp->ccntin > cl->c_cc)
    {
        printf("chread: sleeping for input\n");
        cp->state != CINSLP;      /* sleep until more chars are typed */
        sleep(cl, CPRI);
    }
    splx(priority);

    while (passc(getc(cl)) != -1) /* pass to user space the characters */
        ;                       /* in the clist */
}
chstart(minor_d)
dev_t minor_d;
/*-----
* start io on one of the channels,
* the channel is not busy - guaranteed
* if there is a block on clist
* write output iopb for list
* mark state as busy
*
* note that the block taken off list is put in a the variable: cblock
* so that the ISR may return it to the free list
*-----
*/
{
    struct clist * cl;           /* clist */
    struct cinfo * cp;           /* c information */
    struct ciopb * iopb;         /* iopb filled for output */
    struct cblock * bp;          /* temporary cblock pointer */

    cp = &ch_cinfo[minor_d];
    cl = &cp->clout;
    iopb = &ch_ciopb_io[minor_d].output;
        /* if there are no more cblocks */
    if ((bp = cp->cbout = getcb(cl)) == 0)
        panic("chstart called with nothing to print\n");
```

```

    iopb->func = CFOUTPUT; /* make iopb */
    iopb->chan = minor_d;
    iopb->bufad = &bp->c_data[bp->c_first];
    iopb->buflen = bp->c_last - bp->c_first;

    cp->state != CBUSY; /* mark as busy */

    carb(hp); /* write to mailbox register */
    cmboxwrite(iopb, hp);
    cmboxfree(hp);
}
chintr()
/*-----
 * ISR
 * arbitrate for the mailbox
 * read the mailbox register
 * case(Acknowledged Channel Init CIOPB)
 * validate the iopb
 *
 * case(Acknowledged output)
 * validate the iopb
 * return the cblock that was printed back to the free list
 * wakeup upper half if needed
 * call start to start the next cblock
 *
 * free the mailbox
 *
 *-----
 */
{
    struct ciopb * iopb; /* iopb in the mailbox */
    struct cinfo * cp; /* info structure that caused int */
    struct clist * cl; /* clist pointer */
    int minor_d; /* minor device number */
    char tmp_func; /* temporary function holder */
    int priority; /* priority level */

    carb(hp); /* ask for use of the mailbox */
    cmboxread(iopb, hp); /* read the mailbox */
    /* iopb number (ie channel number) */
    minor_d = (iopb - (struct ciopb *)ch_ciopb_io)/2;
    cp = &ch_cinfo[minor_d]; /* info structure pointer */
    printf("i%x", minor_d);

    switch (iopb->func)
    {
    /*-----*/
    case (CFCHANINIT): /* Acknowledged Channel Init CIOPB */
        /*printf("ci");*/
        if (iopb->stat != CSOPCPL)

```

```

    {
        clogerr(iopb, "channel init failed: operation not complete");
        break;
    }

    /*-- channel initialization wakeup --*/
    cp->state &= CCIWAIT; /* no longer waiting for chan init ack */
    if(cp->state & CCISLP) /* if sleeping for a chan init ack */
    {
        cp->state &= CCISLP; /* no longer sleeping for chan init ack */
        wakeup(cp);         /* wakeup processes waiting for */
    }                       /* chan init ack */

    iopb->func = CFCHARACK; /* from now on it will be input */
    break;

/*-----*/
case (CFCHARACK):          /* character has been typed */
    if(!(iopb->icntrl & CRCVSR))
        pr_iopb(iopb, "char ack but no CRCVSR");
    printf("in char");
    c1 = &cp->c1in;         /* unsolicited input queue */
    putc(iopb->dchar, c1);  /* store the character away */

    cmboxfree(hp);
    carb(hp);              /* ask for use of the mailbox */
    priority = spl0();
    splx(priority);
    cmboxwrite(iopb, hp);  /* write a char ack to the hpsio */
    /*-- input wakeup --*/
    if (cp->state & CINSLP) /* process sleeping on input */
        if (c1->c_cc >= cp->ccntin) /* worth waking up */
        {
            cp->state &= CINSLP;
            wakeup(c1);
        }
    break;

/*-----*/
case (CFOUTPUT):          /* acknowledged output */
    printf("o%x", minor_d);
    if (iopb->stat != CSOPCPL)
    {
        clogerr(iopb, "output failed: operation not complete");
        break;
    }

    /*-- output wakeup --*/
    putcf(cp->cbout);      /* return the cblock to free list */
    if (cp->c1out.c_cc <= CLOWATER) /* wake up top half if drained */
        if (cp->state & COSLP) /* and if sleeping for output */
        {

```



```

        cp->state &= COSLP;
        wakeup(&cp->clout);
    }

    if (cp->clout.c_cc == 0)                /*-- close wakeup --*/
        /* wake close if drained to zero */
    {
        cp->state &= CBUSY;                /* no longer busy */
        if(cp->state & CCLSSLP)            /* and if sleeping for close */
        {
            cp->state &= CCLSSLP;
            wakeup(&cp->clout);
        }
    }
    else
        chstart(minor_d);                /* start up the next transaction */
    break;

default:
    clogerr(iopb, "unexpected interrupt");
    break;
} /* end switch */

cmboxfree(hp);
printf("e");
}

chclose(minor_d)
{
    struct cinfo *cp;                    /* pointer to cinfo structure */
    struct clist *cl;                    /* pointer to clist */
    int priority;                        /* processor priority level */

    cp = &ch_cinfo[minor_d];
    cl = &cp->clout;

    priority = spl_c();
    while (cl->c_cc > 0)                  /* wait for output to drain completely */
    {
        cp->state != CCLSSLP;
        sleep(cl, CPRI);
    }
    cp->state = 0;                        /* reset for the next process to open */
    splx(priority);
}
chioc1()
{
    printf("chioc1:\n");
}
chputc(c, cl)
char c;
struct clist *cl;

```

Chapter 8

```
/*-----
 * put a character onto the clist cl.
 * transformations of single character into multiple chars is also
 * performed
 *-----
 */
{
    switch(c)
    {
        case('\n'):
            chputc('\r', cl);
        default:
            while(putc(c, cl) == -1)
            {
                /* sleep until freelist has blocks */
                printf("cwrite: sleep for cfreelist\n");
                cfreelist.c_flag = 1;
                sleep(&cfreelist, CPRI);
            }
    }
}

/*-----
 * arbitrate for use of hpsio mailbox register
 *-----
 */
carb(hp)
    struct cdev * hp;          /* hpsio device address */
    {
        int i;                 /* counter */

        hp->arb != CO_REQ2;    /* activate the REQ2 bit */
        for (i=0; i<256; ++i) /* delay */
            ;
        while (hp->arb & CO_ACK1) /* wait for board to finish */
            ;
    }

clogerr(iopb, str)
char *str;
struct ciopb * iopb;
/*-----
 * log an error
 *-----
 */
{
    pr_iopb(iopb, str);
}

pr_iopb(p, str)
char *str;
```

```

struct ciopb * p;
/*-----
 * print the contents of an iopb on the console
 *-----
 */
{
    int i;
    printf(str);
    printf("\n iopb: ");
    printf("func %x, stat %x, chan %x, cbuad %x, cmode1 %x, \
        cmode2 %x\n", p->func, p->stat, p->chan, p->cbaud, \
        p->cmode1, p->cmode2);
    printf("cstat %x, icntrl %x, dstat %x, dchar %x, \
        bufad %x, buflen %x\n", p->cstat, p->icntrl, p->dstat, \
        p->dchar, p->bufad, p->buflen);
}

/*-----
 * character device information information for each channel
 *-----
 */
struct cinfo
{
    short          state;          /* state of channel */
    struct cblock  * cbout;        /* cblock being printed */
    struct clist   clout;          /* clist used as output queue */
    struct clist   clin;           /* clist used as unsolicited input queue */
    int ccntin;          /* number of chars needed in input clist */
};

/*-----
 * macros for writing, reading, and freeing the mailbox register
 *-----
 */
#define cmboxwrite(iopb, c) c->mailbox = iopb      /* write to mailbox */
#define cmboxread(iopb, c) iopb = c->mailbox        /* read mailbox */
#define cmboxfree(hp) hp->arb &= CO_REQ2;          /* free the mailbox */

/*-----
 * state bits for controller as a whole as kept in: state
 *-----
 */
#define COPNPRG 0x01          /* open in progress all others must sleep */
#define COPNSLP 0x02          /* sleeping for chance to open */
#define COPENREAD 0x04        /* open for read only */
#define COPENWRITE 0x08       /* open for write only */

/*-----
 * character device state bits (ch_cinfo[x].state)
 *-----

```


Chapter 8

```
*/
#define CCISLP 0x01      /* sleeping for channel init ack */
#define COSLP 0x02      /* sleeping for output to drain */
#define COPEN 0x04      /* channel is ready for output (open) */
#define CBUSY 0x08      /* channel is busy doing output */
#define CCIWAIT 0x10     /* waiting for a chan init ack */
#define CCLSSLP 0x20     /* sleeping for close, output must drain */
#define CINSLP 0x40     /* sleeping for input queue to fill */

#ifndef CPRI
#define CPRI 29          /* defined in lprio.h priority level to sleep */
#endif

/*-----
 * Water marking characteristics.
 *-----
 */
#define CLOWATER 64      /* low water mark on clout */
#define CHIWATER 256     /* hi water mark on clout */
#define CFEW 16          /* characters are added to clout a few */
                        /* at a time. This is a few */

/*-----
 * hpsio firmware command io parameter buffer, one for each channel
 *-----
 */
struct ciopb
{
    unsigned char    func;      /* function code */
    unsigned char    stat;      /* return status */
    unsigned char    chan;      /* channel # */
    unsigned char    cbaud;     /* transmission rate, signal control */
    unsigned char    cmodel;    /* parity type, char length */
    unsigned char    cmode2;    /* #stop bits, Op. mode */
    unsigned char    cstat;     /* line status */
    unsigned char    icntrl;     /* Interrupt Control/SRQ */
    unsigned char    dstat;     /* character status */
    unsigned char    dchar;     /* received character */
    char             * bufad;    /* buffer start address */
    unsigned short    buflen;    /* buffer length */
};

struct ciopb_io
{
    struct ciopb input; /* iopb used for channel init and input */
    struct ciopb output; /* iopb used for output */
};

/*-----
 * HPSIO register description
 *-----
```

```

*/
struct cdev
{
    struct ciopb    *mailbox; /* mailbox register */
    unsigned short  intrvector; /* interrupt vector */
    unsigned short  arb;       /* arbitration / status register */
};

/*-----
 * values for the interrupt vector register of HPSIO (hp->intrvector)
 *-----
*/
#define CINTLEVEL 0x2          /* interrupt level ie sp12() */
#define CVECTOR 0x70          /* interrupt vector number to use */
#define sp1_c() sp12()        /* should be changed with CINTLEVEL */

/*-----
 * values for the arbitration register of HPSIO (hp->arb)
 *-----
*/
#define CREGBASE    (struct cdev *) 0x1f1400 /* base address of registers */
#define CPMCPEND    0x02 /* arbitrator bit */
#define CO_ACK1 0x40 /* arbitrator bit */
#define CO_REQ2 0x80 /* arbitrator bit */

/*-----
 * function codes
 *-----
*/
#define CFCHANINIT 1 /* channel initialization */
#define CFCHARACK 2 /* character acknowledgement */
#define CFCONINT 3 /* configure interrupt */
#define CFOUTPUT 4 /* output */

/*-----
 * returned status codes
 *-----
*/
#define CSOPCPL 0x0 /* operation complete */
#define CSCHECK 0x1 /* cecksum error on download */
#define CSBUFAD 0x2 /* bad BUFAD address */
#define CSMALF 0x3 /* Controller malfunction */
#define CSBUSERR 0x4 /* HPSIO Bus Error */
#define CSFUNC 0x5 /* Bad Func code */
#define CSBUFLEN 0x6 /* bad BUFLen count */
#define CSTIME 0x7 /* Time out before Tx ready */
#define CSPARITY 0x8 /* HPSIO parity error */
#define CSINT 0x9 /* HPSIO processor unexpected interrupt */
#define CSDUART 0xa /* Unexpected DUART interrupt */
#define CSADDR 0xb /* HPSIO processor address error */

```

Chapter 8

```
#define CSSERVREQ 0xc      /* service request */
#define CSCHAN    0xd      /* bad CHAN number */
#define CSCHANINIT 0xe     /* invalid function till channel init */
#define CSDOWN    0xf      /* download in progress */

/*-----
 * values of icntrl that are needed
 *-----
 */
#define CRCVSR      0x04    /* received character interrupt */

/*-----
 * other stuff thats needed
 *-----
 */
#ifndef TRUE
#  define TRUE 1
#endif
#ifndef FALSE
#  define FALSE 0
#endif
#ifndef min
#  define min(a,b)      (a>b ? b : a)
#endif
```


Chapter 9.

A TTY Driver

INTRODUCTION

The Device Independent Kernel is the software layer above the device driver.

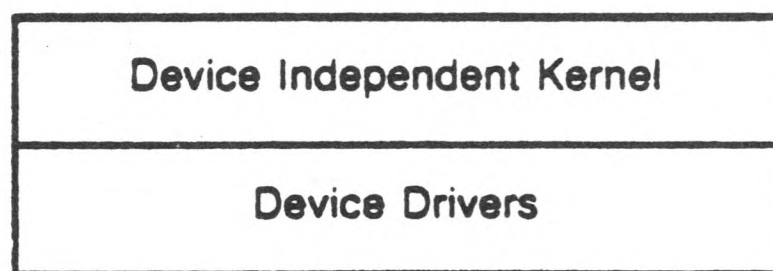


Figure 1. Kernel Layers

The kernel contains a layer of software, called Line Discipline Routines (LDR), that simplify the creation of terminal device drivers. The goal of both the LDR and the device driver is to provide the functionality as described in *termio(7)* in the *Administrator Reference*, UP-11761.

The device driver is concerned with low-level device-specific functions, and the LDR are concerned with high-level device-independent functions.

The Device Independent Kernel communicates with the device driver through the LDR. The LDR and the driver functions are closely related, but the driver is

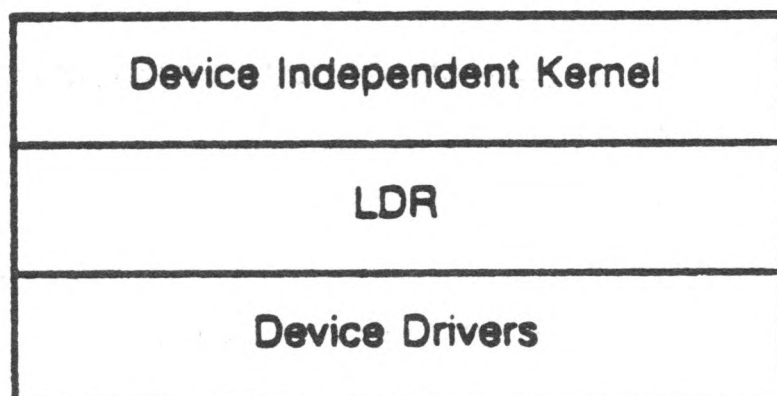


Figure 2. Kernel Layers Including LDR

primarily concerned with handling interrupts and setting terminal characteristics.

A user can write an LDR and substitute it into the kernel with no change to the existing drivers.

EXAMPLE OF LDR FUNCTIONS

Here are some LDR functions:

- Echo characters entered on a terminal keyboard.
- Input character processing.
- Delete character processing.
- Delete line processing.
- Signal handling
- Process suspension until a specific time limit has expired, or a character is entered, or a line is entered.
- New line character interpretation.
- Delay timing insertion for older devices.

A general description of tty drivers is provided in the following sections. The NEC driver, which controls port A and port B on the PMC board, is described to provide specific device driver examples.

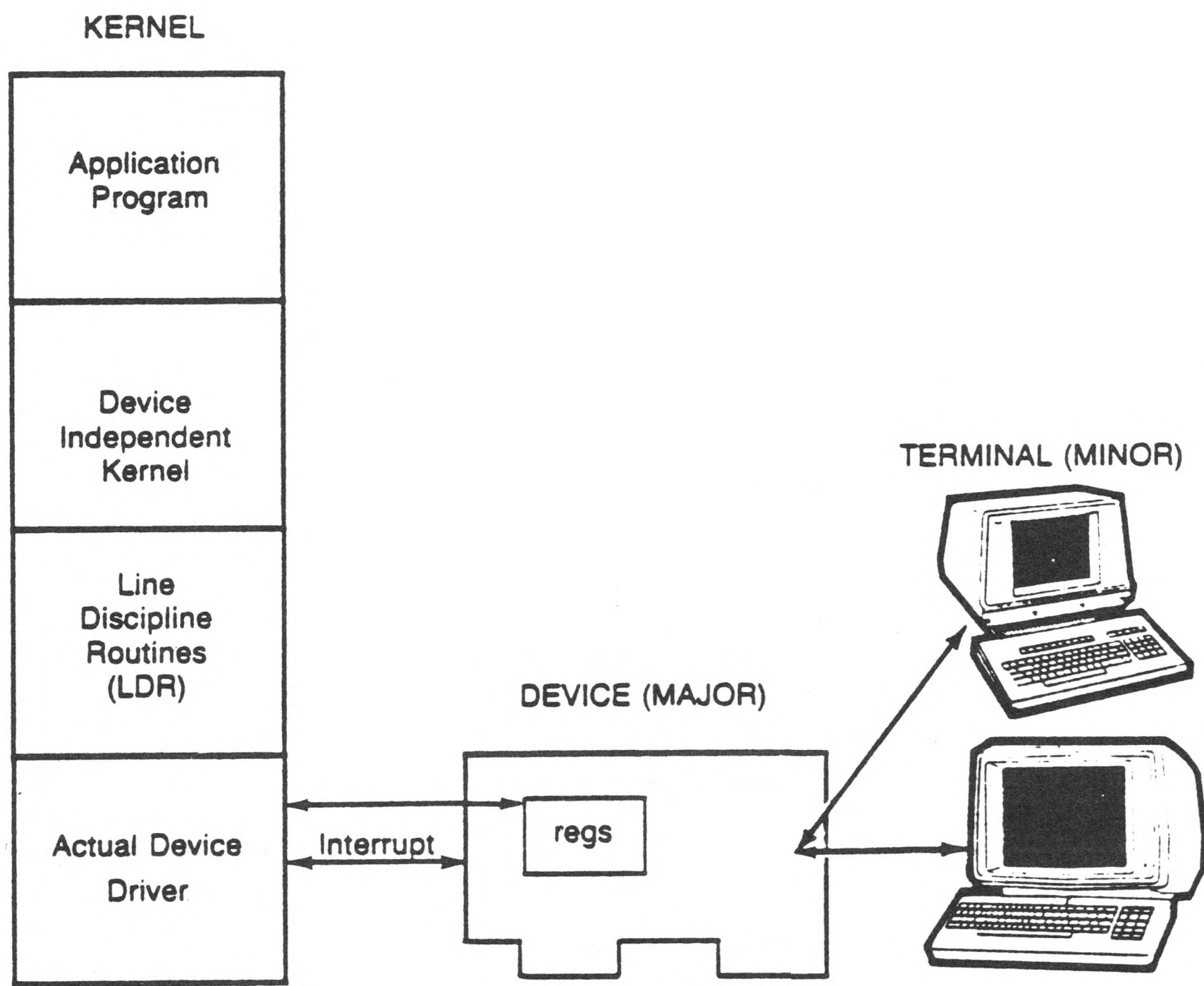


Figure 3. Relationship Between Software and Hardware

NOTE: The *regs* box in the device (Figure 3) is a set of registers used by the driver to control the device.

GENERAL CONTROL FLOW

Initialization is performed to set up hardware parameters. Codes are written to control registers to set characteristics such as parity, bits per character, interrupt initialization, etc. The following lists identify input and output functions.

Input:

- The user enters a character and the terminal transmits that character as a serial stream of bits.
- The bits are received by the device and assembled into a character.
- The character is examined for validity by the device.
- The character is put into the receive buffer register.
- An interrupt is generated.
- The receive Interrupt Service Routine (ISR) in the driver is invoked.

Output:

- When the transmitter portion of the device is on, and a character has been placed in the transmit buffer register by the driver, the character is converted to a stream of bits and sent to the terminal.
- When the transmit buffer register becomes empty, an interrupt is generated.
- The transmit ISR in the driver is invoked to write another character to the device.

NEC TERMINAL CONTROLLER

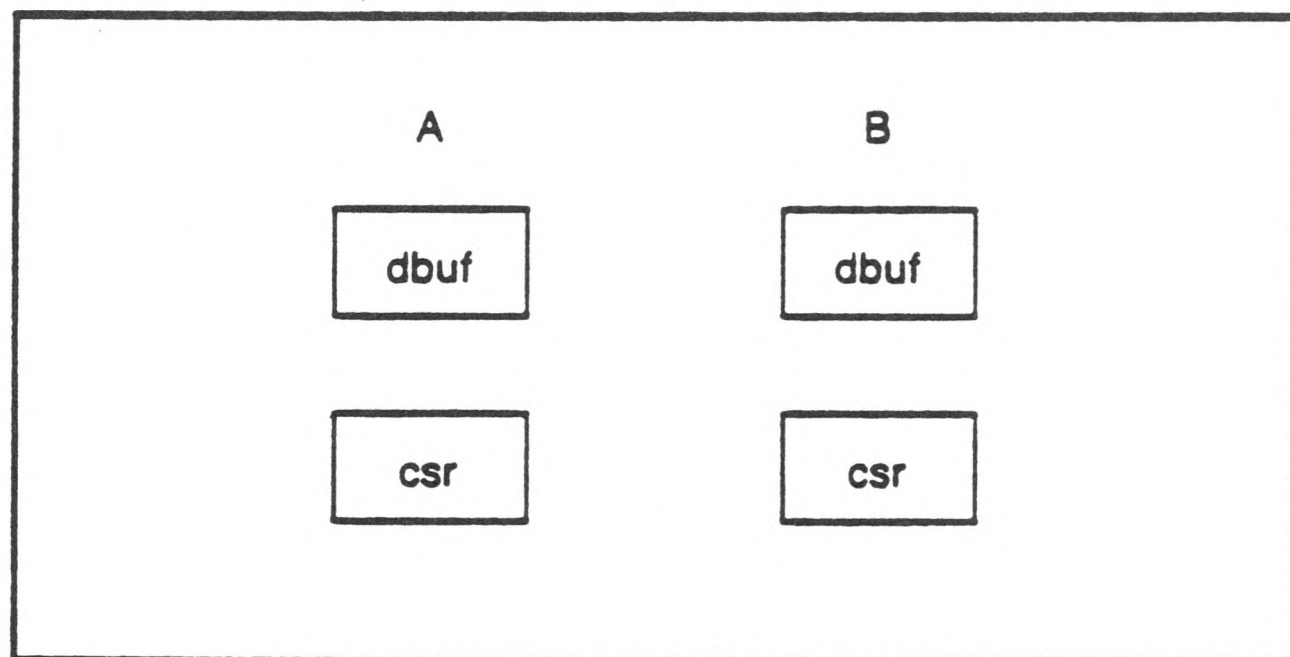


Figure 4. NEC Chip Block Diagram

Figure 4 shows that the NEC chip has two registers associated with each channel. Reading or writing these registers controls the NEC chip in this way:

- Reading from a *dbuf* (data buffer register) returns the value of the oldest character stored in the chip's three character *fifo* queue.
- Writing to a *dbuf* causes a character to be transmitted to the terminal.
- To obtain status information about the chip, the driver must read from a *csr* (control status register).
- To control the hardware parameters of the chip, the driver must write to a *csr* (control status register).

This C declaration defines the device:

```
struct device
{
    char dbuf;
    char dum1;
    char csr;
    char dum2;
}
struct device *nec_addr = (struct device*)0xe00101;
```

To read a character from port A, execute this C statement:

```
ch = nec_addr->dbuf;
```

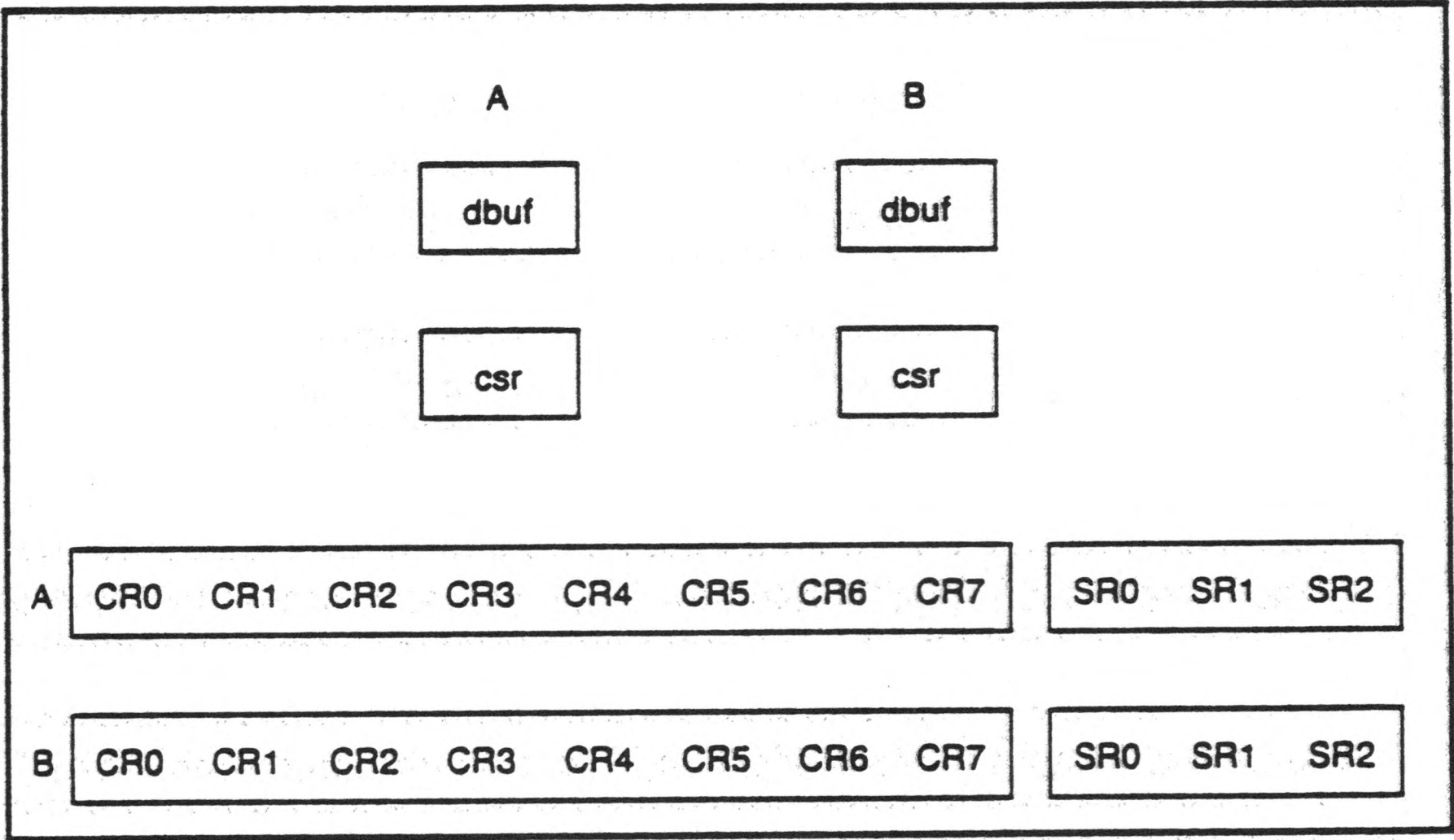


Figure 5. NEC Chip Registers

The control status register (*csr*) is used to access eight internal control registers and three internal status registers. By default, control/status register zero is manipulated. The lower three significant bits in control register zero are used as a register pointer. The following C statements writes configuration information to control register two:

```
#define RP2 2
#define NON_VECTOR 0

nec_addr->csr = RP2;
nec_addr->csr = NON_VECTOR;
```

NOTE: The remaining details of the NEC chip are described as required.

DATA FLOW DIAGRAM

Figure 6 describes the data flow for the NEC driver.

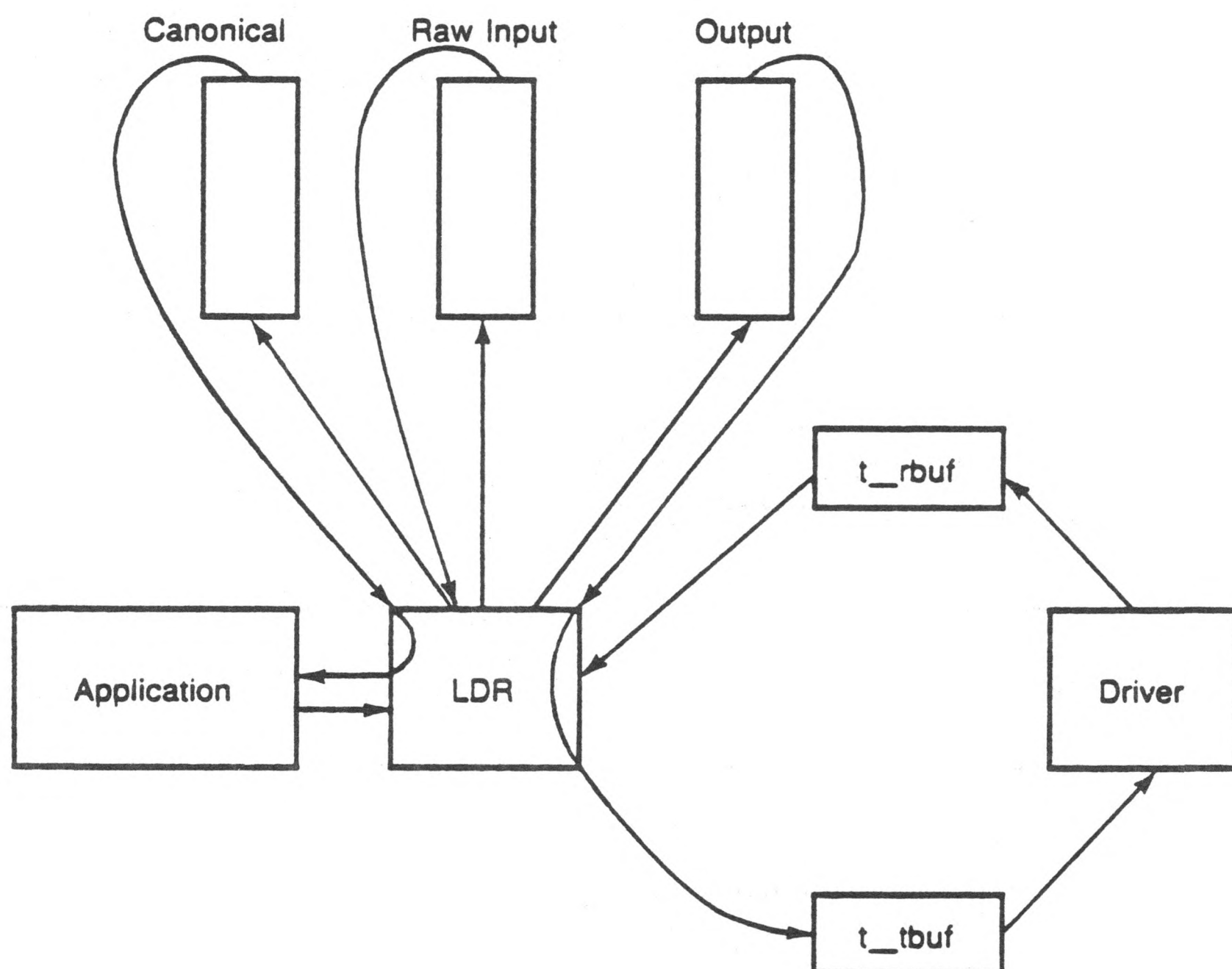


Figure 6. NEC Driver Data Flow

NOTE: The canonical input queue, raw input queue, and output queue are all controlled by the LDR. The queues are not accessed by the device driver.

The *t_rbuf* and *t_tbuf* are the receive character buffer and transmit character buffer. These buffers are used to pass character buffers between the driver and the LDR. For example, *t_rbuf* is partially filled by the receive ISR of the device driver. An LDR function is called to process the characters in the buffer.

Control Flow

Primary events, related driver and LDR functions are shown in the following chart.

ENTRY POINT	DRIVER FUNCTION	LDR FUNCTION
receive int.	necintr-receiver	l_input
transmit int.	necintr-transmitter	l_output
upper layers	necopen	l_open t_topen
upper layers	necclose	l_close
upper layers	necread	l_read
upper layers	necwrite	l_write
upper layers	necioctl	l_ioctl
upper layers	not used	l_mdminit

Figure 7. Driver/LDR Functions

The kernel is configured so that *l_read* and *l_write* are called directly and *necread* and *necwrite* are never called.

LINE SWITCH TABLE

The line switch table defines entry points into the LDR and is similar in structure to the block device switch table and the character device switch table. The line switch table is defined in */usr/include/sys/conf.h*.

```
/*
 * Line discipline switch.
 */
struct linesw {
    int (*l_open)();    /* on open to a minor device */
    int (*l_close)();   /* after all closes to a minor device */
    int (*l_read)();    /* on a read system call */
    int (*l_write)();   /* on a write system call */
    int (*l_ioctl)();   /* on a ioctl system call */
    int (*l_input)();   /* from driver's receiver ISR */
    int (*l_output)();  /* from driver's transmitter ISR */
    int (*l_mdmint)();  /* from driver on modem status change */
};
extern struct linesw linesw[];

extern int linecnt;
```

This data structure permits multiple LDRs to exist in the kernel, and each terminal or application can choose which LDR to use.

TTY STRUCTURE

A tty structure exists for each terminal (minor device) in the system. All data needed to monitor the terminal status are kept the tty structure. The tty structure is defined in */usr/include/sys/tty.h*.

```
struct ccblock {
    caddr_t c_ptr;    /* buffer address */
    ushort c_count;   /* character count */
};
```

```

        ushort c_size;    /* buffer size */
    };

#define NCC 8
struct tty {
    struct clist t_rawq;    /* raw input queue */
    struct clist t_canq;    /* canonical queue */
    struct clist t_outq;    /* output queue */
    struct ccblock t_tbuf; /* tx control block */
    struct ccblock t_rbuf; /* rx control block */
    int (* t_proc)();      /* routine for device functions */
    ushort t_iflag;        /* input modes */
    ushort t_oflag;        /* output modes */
    ushort t_cflag;        /* control modes */
    ushort t_lflag;        /* line discipline modes */
    long t_state;          /* internal state */
    short t_pgrp;          /* process group pid */
    short t_delct;         /* delimiter count */
    char t_line;           /* line discipline */
    char t_term;           /* terminal type */
    char t_tmflag;         /* terminal flags */
    char t_col;            /* current column */
    char t_row;            /* current row */
    char t_vrow;           /* variable row */
    char t_lrow;           /* last physical row */
    char t_hqcnt;          /* no. high queue packets on t_outq */
    char t_dstat;          /* used by terminal handlers
                           and line disciplines */
    unsigned char t_cc[NCC]; /* settable control chars */
};

```

The fields in the `tty` structure used by the device driver are described below. The remaining fields are used only by the LDR.

t_tbuf

Transmit buffer. This buffer is filled by the LDR and emptied by the driver. The LDR function `l_output` fills this buffer.

t_rbuf

Receive buffer. This buffer is filled by the driver. The LDR function `l_input` processes this buffer.

t_proc

Driver procedure. This identifies a function that can be called by the LDR to change the state of the driver.

t_iflag

Input mode bit flags. Specifies input control.

These flags are used by the driver:

IXON

XON/XOFF output flow control.

IXOFF

XON/XOFF input flow control

IXANY

Use any character to start output flow, not just <control q>.

IRTS

Use the request-to-send signal to start and stop the input flow from the terminal.

PARMRK

Mark parity framing and overrun errors.

IGNBRK

Ignore break characters.

BRKINT

Send a signal to all processes associated with the terminal when a break character is received.

INPCK

Enable the use of parity checking for input and output characters.

IGNPAR

Ignore parity on input. This permits output parity to be generated while ignoring input parity errors.

ISTRIP

Strip off most significant bit of eight bit character.

ERR_BEL

Transmit a bell on errors.

t_cflag

Control bit flags. Used to specify the hardware configuration of the device.

t_state

State of the terminal device bit flags.

These flags are used by the driver:

WOPEN

Waiting to open. The open function of the driver has been called but it isn't open yet. It may be suspended, waiting for a terminal or modem to be turned on.

ISOPEN

The terminal is open.

CARR_ON

The carrier was detected (ON).

BUSY

The driver is busy performing output. When in this state, an interrupt occurs after the character is transmitted and the device is unable to accept more characters to transmit.

OASLP

The LDR routines are asleep waiting for output to complete.

TTXON

A XON character needs to be transmitted by the transmit interrupt routine.

TTXOFF

A XOFF character needs to be transmitted by the transmit interrupt routine.

TIMEOUT

The device is busy for a specific period of time. No characters should be transmitted when this flag is on.

TTSTOP

Output is stopped (suspended). No characters should be transmitted when this flag is on.

TBLOCK

Input is stopped (blocked).

t_line

Line discipline identification. A number corresponding to the line discipline routines being used. This is an index into the *linesw* table.

RS-232-C HARDWARE SIGNALS: DTR AND DCD

RS-232-C is an industry standard that describes the interface between a computer and a modem (DTE to DCE). A small subset of this standard is used by the NEC chip. The device driver can set or sense certain pins in the RS-232 connector between the device controller and the terminal. The DTR and DCD signals show existence of a device in two ways:

- DTR (Data Terminal Ready) notifies the terminal that the computer is turned on. DTR is enabled at the first open of the minor device and remains ON.
- DCD (Data Carrier Detect) notifies the computer that the terminal is turned on. Normally, DCD is enabled when the terminal power switch is turned on.

NECOPEN DEFINITION

The open routine performs these functions:

- The internal representation of carrier is turned on or off.
- The routine sleeps until carrier is detected, unless this option is overridden by the open flag.
- The LDR *l_open* routine is called.
- The first open for a device executes the LDR, *ttinit*, to set up flag default hardware configuration values.

NECOPEN LISTING

```
/*-----  
* pnec.open.c -- open a terminal.  
*-----  
*/
```



```

void necopen(minor_d, flag)
dev_t minor_d; /* minor device number */
int flag;      /* open flag see fopen.h */
{
    register struct tty *tp; /* nec tty to initialize */
    struct device *rp = nec_addr + minor_d; /* nec register pointer */
    int priority; /* previous priority */
    extern int necproc(); /* state changes by LDR */

    if ( minor_d >= nec_cnt ) /* minor number within range */
    {
        u.u_error = ENXIO;
        return;
    }
    necsave("necopen 2");
    tp = &nec_tty[minor_d]; /* initialize tty pointer */

    if ((tp->t_state & (ISOPEN | WOPEN)) == 0) /* first open for device */
    {
        ttinit(tp); /* set default termio flags */
        tp->t_proc = necproc; /* LDR needs state change proc */
        necparam(minor_d); /* initialize HW as set by ttinit */
    }
    if( rp->csr & DCD ) /* if DCD then we're ready */
        tp->t_state |= CARR_ON; /* set carrier on flag */
    else
        tp->t_state &= CARR_ON; /* carrier is not on */

    if ( !(flag & FNDELAY) ) /* the user wants to wait */
    {
        /* until carrier is detected */
        priority = spl3();
        while( (tp->t_state & CARR_ON) == 0 ) /* until carrier is on */
        {
            tp->t_state != WOPEN; /* we are waiting to open */
            sleep((caddr_t)&tp->t_canq, TTIPRI); /* and sleeping */
        }
        splx(priority);
    }
    /* set up process groups. mark terminal as ISOPEN. set up t_rbuf. */
    (*linesw[tp->t_line].l_open)(tp); /* call necproc(tp, T_INPUT) */
}

```

DEVICE DRIVER INTERRUPT SERVICE ROUTINES

Two interrupt service routines (ISR) are used to handle interrupts generated by a device.

Receive interrupt service routine:

- Reads the status register to check for errors (framing, overrun, or parity).
- Reads the receive buffer register.
- Resets interrupts.
- If *t_rbuf* has been initialized, puts the character read into *t_rbuf* and calls the LDR input interrupt function.

Transmit interrupt service routine:

- Resets transmit interrupts.
- If there are characters in *t_tbuf*, takes the next character from *t_tbuf*.
- Writes the character to the device transmit buffer register or calls the LDR function to put more characters in *t_tbuf*.

T_rbuf

The receive buffer is filled by the device driver and passed to the *l_input* routine of the LDR for processing.

```
struct ccblock {  
    caddr_t  c_ptr;           /* buffer address */  
    ushort  c_count; /* character count */  
    ushort  c_size;          /* buffer size      */  
};
```

WHERE:

c_ptr

Pointer to a buffer where received characters can be placed.

c_count

Amount of space left in buffer. Initially the same as *c_size*. Decrement each time a character is added to the *t_rbuf*.

c_size

Size of the buffer. Never changed by the driver.

T_rbuf can be in one of four states:

Uninitialized state

The LDR routine *l_input* needs to be called to initialize *t_rbuf*. *C_ptr* is 0 when in this state. All received characters should be ignored when in this state.

Empty state

When in this state the driver has not started filling *t_rbuf*. *C_ptr* has been initialized by the LDR to point to an empty buffer. *C_size* and *c_count* are initially both the same value (*CLSIZE*).

Filling state

When in this state, the driver is in the process of filling the *t_rbuf*. Each character is put into the buffer. *C_count* needs to be decremented at the same time.

Full state

When the buffer becomes full or when the receive ISR has read all characters for a given interrupt, the buffer pointer is reset and the LDR *l_input* routine is called.

T_tbuf

The transmit buffer is filled by the LDR *l_output* routine and is passed to the device driver for writing to the device.

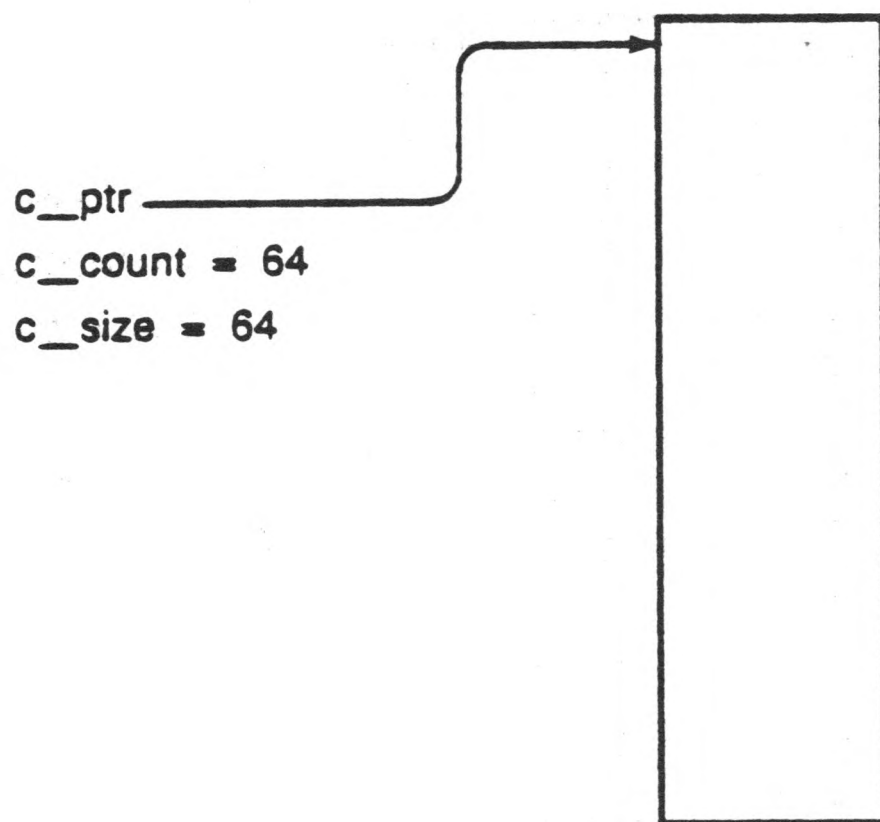


Figure 8. Buffer Empty

```
struct ccblock {  
    caddr_t  c_ptr;      /* buffer address */  
    ushort   c_count; /* character count */  
    ushort   c_size;     /* buffer size */  
};
```

WHERE:

c_ptr

Pointer to a buffer of characters.

c_count

Number of characters left in buffer. Initially the same as *c_size*. Decrement each time a character is transmitted.

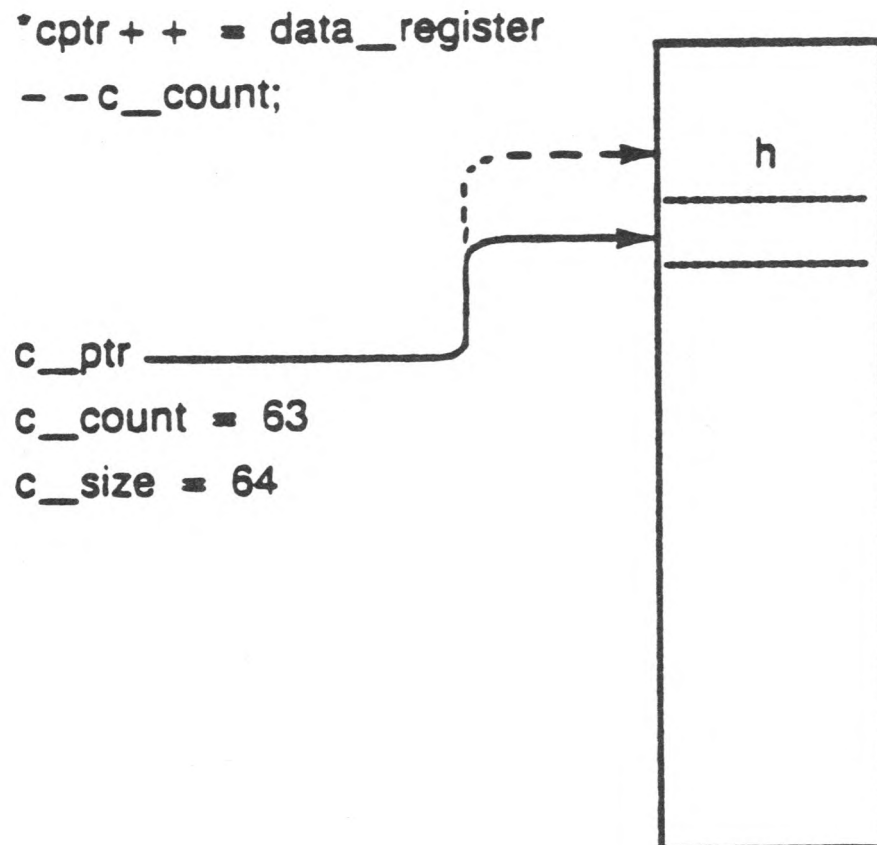


Figure 9. Buffer Filling

c_size

Size of the buffer. Never changed by the driver.

T_tbuf can be in one of four states.

Uninitialized state

The line discipline routine *l_output* needs to be called to initialize *t_tbuf*. *C_ptr* is 0 or *c_size* is 0 when in this state.

Full state

When in this state the driver has not started emptying *t_tbuf*. *C_ptr* has been initialized by the LDR to point to a buffer. *C_size* and *c_count* are initially both the size of the buffer.

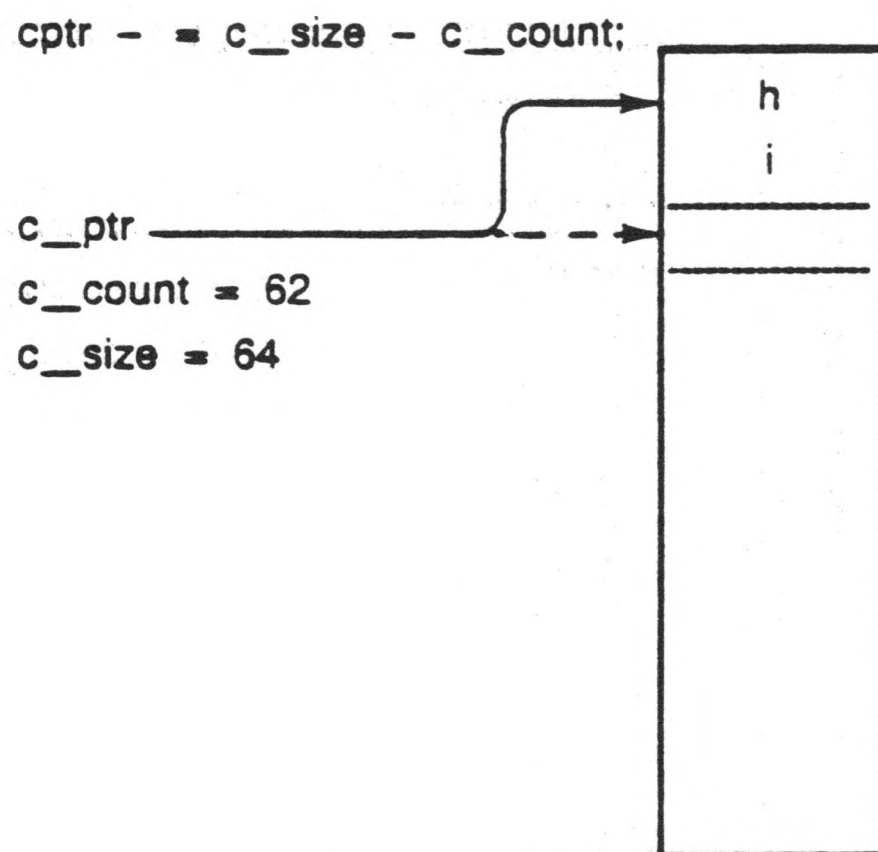


Figure 10. All Characters Recieved

Transmitting state

When in this state, the driver is in the process of transmitting the *t_tbuf*.

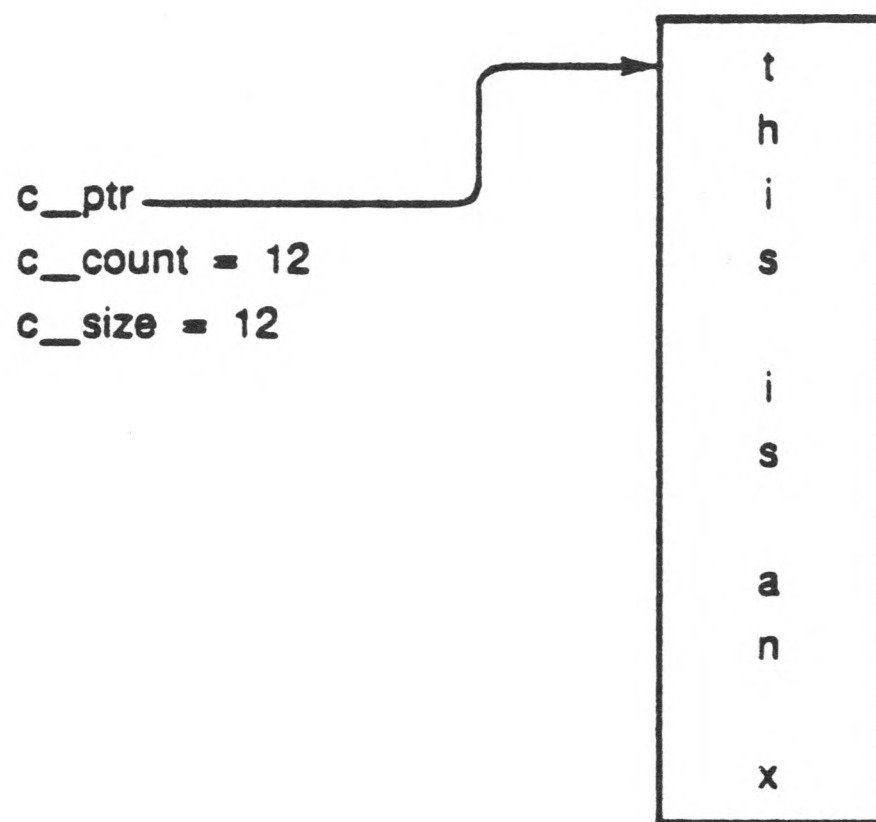


Figure 11. Buffer Initialized

```
data_register = *cptr + +;  
-- c_count;
```

```
c_ptr  
c_count = 9  
c_size = 12
```

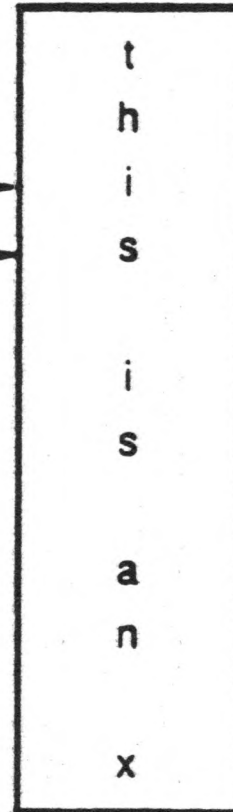


Figure 12. Buffer Transmitting

Empty state

When the buffer becomes empty, the *c_ptr* value is reset and the *l_output* LDR routine is called to get another buffer.

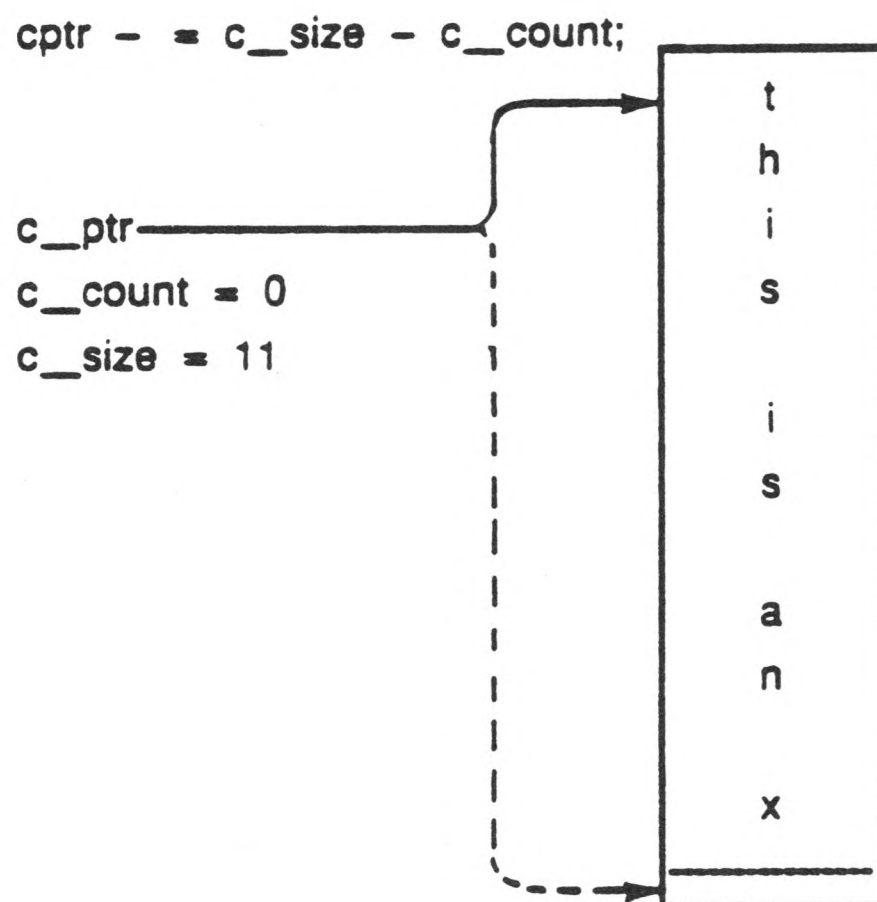


Figure 13. Buffer Empty

RECEIVER/TRANSMITTER LISTING

```

/* Variable Definition
 * rp - register pointer for the port
 * nec_addr - register pointer to port A
 * tp - pointer to tty structure for the interrupting terminal
 */

/* Receiver interrupt service routine */

rp->csr = RP1;          /* check for errors */
if ( rp->csr & SP_FLAGS ) /* Framing, Overrun, Parity */
    goto SRCOND ;

c = rp->dbuf;           /* read a character */

rp->csr = RINT_RESET;   /* reset interrupts */

```



```
nec_addr->csr = END_INT ;

if ( tp->t_rbuf.c_ptr == NULL ) /* why not check for c_count == 0 */
    continue ;

*tp->t_rbuf.c_ptr++ = c ; /* stash character in t_rbuf */
--tp->t_rbuf.c_count ;
                                /* adjust c_ptr correctly */
tp->t_rbuf.c_ptr -= tp->t_rbuf.c_size - tp->t_rbuf.c_count ;
(*linesw[tp->t_line].l_input)(tp) ; /* LDR input */

/* Transmitter interrupt service routine */

rp->csr = PENDING_XINT_RESET ; /* reset interrupts */
nec_addr->csr = END_INT ;      /* Done with interrupt */

tbuf = &tp->t_tbuf ;           /* initialize to transmit buffer */
if( tbuf->c_ptr != NULL && tbuf->c_count != 0 )
{
    /* if there is something to xmit */
    tbuf->c_count-- ;           /* write character */
    rp->dbuf = *tbuf->c_ptr++ ;
}
else
{
    tp->t_state &= BUSY ;       /* not busy doing output */
    necproc(tp, T_OUTPUT) ;    /* get another t_buf and prime the pump */
                                /* by writing the first character */
}
```

INPUT/OUTPUT FLOW CONTROL

If two devices are communicating at different or variable band rates, the devices must have a method to start and stop each other. Start/stop control of a device is called flow control. For example, if a computer sends characters to a printer at speeds faster than the printer can actually print, the printer stores the characters in a local buffer. When the local buffer is almost full, the printer must stop the flow of characters so that no characters are lost.

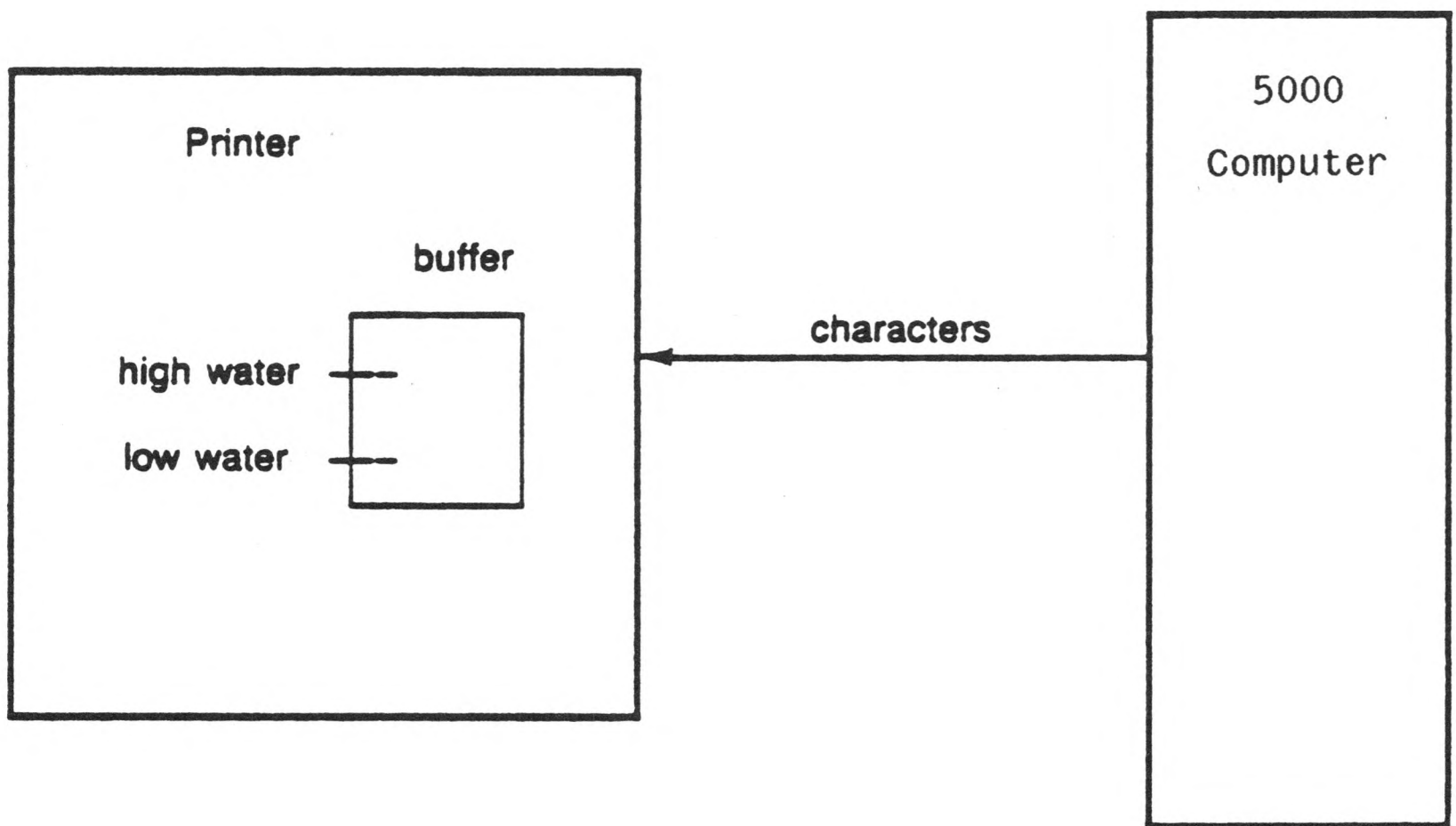


Figure 14. Flow Control

The two common methods of handling I/O flow control are:

- XON/XOFF flow control
- Ready/Busy flow control

XON/XOFF Flow Control

The XON/XOFF protocol uses reserved characters within the data stream that have special meaning and are not data characters: XOFF is a stop character and XON is a start character.

Characters are transmitted to the printer at a rate faster than the printer can print. The printer stores characters in its internal (local)

buffer. When the buffer reaches a high water mark, the printer transmits an XOFF character to the computer. The computer must stop transmitting characters. The printer continues to print the characters in the buffer until a low water mark is reached, at which time the printer transmits a XON character. The computer can then resume sending characters.

XON/XOFF Protocol for Output Flow Control

The output flow of characters from the computer to the printer is controlled by the printer. The device driver handles output flow control.

The *t_iflag* in the *tty* structure has a bit flag which shows that XON/XOFF output flow control is being used. If (*t_iflag* & IXON) is true, the protocol is active.

The routine that detects a change in the state of output flow control is the receive ISR. The receive ISR algorithm has to be modified like this:

- Read the status register to check for errors (framing, overrun, or parity).
- Read the receive buffer register.
- Reset interrupts.
- If the character is a XOFF character, SUSPEND output.
- If the character is a XON character, RESUME output.
- If *t_rbuf* has been initialized, put the character read into *t_rbuf* and call the LDR input interrupt function.

RECEIVE ISR LISTING

```

while(rp->csr & CHAR_AVAILABLE) /* while characters to receive */
{
    rp->csr = RP1; /* check for errors */
    if ( rp->csr & SP_FLAGS ) /* Framing, Overrun, Parity */
        goto SRCOND ;

    c = rp->dbuf; /* read a character */

    rp->csr = RINT_RESET; /* reset interrupts */
    nec_addr->csr = END_INT ;

/* begin XON/XOFF output flow control section */
    if (tp->t_iflag & IXON)
    {
        register char ctmp; /* may be an XON or XOFF char */
        ctmp = c & 0177;
        if (tp->t_state & TTSTOP) /* output is suspended */
        { /* resume on start character */
            if (ctmp == CSTART || tp->t_iflag & IXANY)
                (*tp->t_proc)(tp, T_RESUME);
        }
        else
        {
            if (ctmp == CSTOP) /* stop character received */
                nec_addr->csr = NON_VECTOR;
            (*tp->t_proc)(tp, T_SUSPEND); /* suspend output */
        }
        /* start/stop chars go no further */
        if (ctmp == CSTART || ctmp == CSTOP)
            continue ;
    }
/* end XON/XOFF output flow control section */

    /* stash character in r_buf */
    if ( tp->t_rbuf.c_ptr == NULL ) /* why not check for c_count == 0 */
        continue ;

    *tp->t_rbuf.c_ptr++ = c ;
    --tp->t_rbuf.c_count ;

    /* adjust c_ptr correctly */
    tp->t_rbuf.c_ptr -= tp->t_rbuf.c_size - tp->t_rbuf.c_count;
    (*linesw[tp->t_line].l_input)(tp); /* LDR input */
}
break;

```

XON/XOFF PROTOCOL FOR INPUT FLOW CONTROL

The computer also controls the flow of input characters from fast peripherals using the XON/XOFF protocol, if $(t_iflag \& IXOFF)$ is true.

When the computer wants to stop the peripheral from sending characters, it transmits an XOFF character. This function is called BLOCKing input. When the computer is capable of receiving characters, it transmits an XON character. This function is called UNBLOCKing input. The BLOCKing and UNBLOCKing state is detected by the LDR, which uses a water marking scheme on the raw input queue.

The state transition could occur when the device is busy. If the device is busy, the t_state flags TTXON or TTXOFF are set, indicating that an XON or an XOFF needs to be sent at the next invocation of the transmit ISR.

The transmit interrupt service routine algorithm includes these functions:

- Reset transmit interrupts.
- Write an XON to the transmit buffer register if the TTXON flag is set.
- Write an XOFF to the transmit buffer register if the TTXOFF flag is set.
- If there are characters in t_tbuf :
 1. Take the next character from t_tbuf .
 2. Write the character to the device transmit buffer register.
 3. Else call the LDR function to put more characters in t_tbuf .

TRANSMITTER ISR LISTING

```

/*-----
 * pnec.xint.c -- transmit interrupt handler, include file
 * on input. Interrupts are turned off. T_state is marked as BUSY.
 * rp is pointing to the correct register set for a or b.
 *-----
 */
rp->csr = PENDING_XINT_RESET ; /* reset interrupts */
nec_addr->csr = END_INT ;      /* Done with interrupt */

sysinfo.xmtint++ ;             /* increment tallies in sysinfo */
check_tally( chan, T_OUTS ) ;

if ( tp->t_state & TTXON )      /* If waiting to transmit */
{                               /* XON state */
    tp->t_state &= TTXON ;      /* clear XON state */
    rp->dbuf = nec_xon ;       /* write a XON character */
}
else if ( tp->t_state & TTXOFF ) /* If waiting to transmit */
{                               /* XOFF state */
    tp->t_state &= TTXOFF ;     /* clear XOFF state */
    rp->dbuf = nec_xoff ;      /* write a XOFF character */
}
else                           /* normal character can be output */
{
    register struct ccblock *tbuf ; /* point to transmit buffer */

    if( tp->t_state & TTSTOP ) /* output is suspended <ctrl s> */
    {
        tp->t_state &= BUSY ; /* not busy doing output */
        continue ;
    }

    tbuf = &tp->t_tbuf ;      /* initialize to transmit buffer */
    if( tbuf->c_ptr != NULL && tbuf->c_count != 0 )
    {
        /* if there is something to xmit */
        tbuf->c_count-- ;      /* write character */
        rp->dbuf = *tbuf->c_ptr++ ;
    }
    else
    {
        tp->t_state &= BUSY ; /* not busy doing output */
        necproc(tp, T_OUTPUT) ; /* get another t_buf and prime the pump */
        /* by writing the first character */
    }
}
break ;

```


READY/BUSY FLOW CONTROL

The ready/busy flow control protocol is similar to XON/XOFF. RS-232C signals are used to control the flow of characters. The Request-To-Send (RTS) pin of the NEC controller is controlled by the device driver and can be used by the device driver for input flow control.

- When RTS is enabled, the terminal can transmit characters
- When RTS is disabled, the terminal cannot transmit characters.

The peripheral device uses a signal electrically connected to the clear to send (CTS) pin on the NEC controller. The driver doesn't participate in output flow control.

- When CTS is enabled, characters can be transmitted by the NEC controller.
- When CTS is disabled, characters can't be transmitted by the NEC controller.

The NEC controller stops transmitting characters when the CTS pin is disabled and stops generating interrupts. Therefore, the device driver does not participate in ready/busy input flow control.

PARITY

Incoming characters may have a parity bit appended to the input character. If parity is enabled, the receiver ISR must clear this bit before storing it in the *t_rbuf*.

The driver can be put in a mode (*t_iflag* & PARMRK) where characters received with parity, framing, or overrun errors are placed in *t_rbuf*, as shown in this three character sequence:

0377 0 x

NOTE: x is the character received in error.

If the receiver ISR gets a natural 0377 character, the 0377 should be substituted with the two character sequence 0377 0377.

RECEIVER ISR LISTING

```

/*-----
 * pnec.rint.c -- receive isr, interrupts turned off, may or may
 * not be a place to put the characters
 *-----
 */
sysinfo.rcvint++ ;          /* increment tallies in sysinfo */
while(rp->csr & CHAR_AVAILABLE) /* while characters to receive */
{
    rp->csr = RP1;          /* check for errors */
    if ( rp->csr & SP_FLAGS ) /* Framing, Overrun, Parity */
        goto SRCOND ;

    c = rp->dbuf;          /* read a character */

    rp->csr = RINT_RESET;   /* reset interrupts */
    nec_addr->csr = END_INT ;
    check_tally(chan, T_INPUTS);

    if( !(tp->t_state & (ISOPEN|WOPEN)) ) /* if not open(ing) forget it */
        continue ;

    /* begin XON/XOFF output flow control section */
    if (tp->t_iflag & IXON)
    {
        register char ctmp;          /* may be an XON or XOFF char */
        ctmp = c & 0177;
        if (tp->t_state & TTSTOP)      /* output is suspended */
        {                             /* resume on start character */
            if (ctmp == CSTART || tp->t_iflag & IXANY)
                (*tp->t_proc)(tp, T_RESUME) ;
        }
    }
    else
    {

```

```

        if (ctmp == CSTOP)          /* stop character received */
            (*tp->t_proc)(tp, T_SUSPEND) ; /* suspend output */
        }
        /* start/stop chars go no further */
        if (ctmp == CSTART || ctmp == CSTOP)
            continue ;
    }
/* end XON/XOFF output flow control section */

        /* stash character in t_rbuf */
        if ( tp->t_rbuf.c_ptr == NULL ) /* why not check for c_count == 0 */
            continue ;

        if (tp->t_cflag & PARENB) /* strip off parity bit(s) if parity enabled */
        {
            switch (tp->t_cflag & CSIZE) /* number of bits in character */
            {
                case CS7:
                    c &= 0x7f;
                    break;
                case CS6:
                    c &= 0x3f;
                    break;
                case CS5:
                    c &= 0x1f;
                    break;
            }
        }

        *tp->t_rbuf.c_ptr++ = c ;
        --tp->t_rbuf.c_count ;

        if ( c == 0377 && tp->t_iflag&PARMRK ) /* 0377 is 0377 0377 */
        {
            /* this avoids ambiguity */
            *tp->t_rbuf.c_ptr++ = 0377 ; /* when parity errors are marked */
            --tp->t_rbuf.c_count ; /* and passed to the LDR */
        }

        /* adjust c_ptr correctly */
        tp->t_rbuf.c_ptr -= tp->t_rbuf.c_size - tp->t_rbuf.c_count;
        (*linesw[tp->t_line].l_input)(tp) ; /* LDR input */
    }
    break;

```

ISTRIP

The last addition to the receiver interrupt routine is the ISTRIP input option. If ISTRIP is set, valid input characters are first stripped to 7 bits; otherwise, all 8

bits are processed.

RECEIVER ISR -- FINAL VERSION

```

/*-----
 * pnec.rint.c -- receive isr, interrupts turned off, may or may
 * not be a place to put the characters
 *-----
 */
sysinfo.rcvint++ ;          /* increment tallies in sysinfo */
while(rp->csr & CHAR_AVAILABLE) /* while characters to receive */
{
    rp->csr = RP1;          /* check for errors */
    if ( rp->csr & SP_FLAGS ) /* Framing, Overrun, Parity */
        goto SRCOND ;

    c = rp->dbuf ;          /* read a character */

    rp->csr = RINT_RESET;    /* reset interrupts */
    nec_addr->csr = END_INT ;
    check_tally(chan, T_INPUTS);

    if( !(tp->t_state & (ISOPEN|WOPEN)) ) /* if not open(ing) forget it */
        continue ;

/* begin XON/XOFF output flow control section */
if (tp->t_iflag & IXON)
{
    register char ctmp;      /* may be an XON or XOFF char */
    ctmp = c & 0177;
    if (tp->t_state & TTSTOP) /* output is suspended */
    {
        /* resume on start character */
        if (ctmp == CSTART || tp->t_iflag & IXANY)
            (*tp->t_proc)(tp, T_RESUME) ;
    }
    else
    {
        if (ctmp == CSTOP) /* stop character received */
            (*tp->t_proc)(tp, T_SUSPEND) ; /* suspend output */
    }

    /* start/stop chars go no further */
    if (ctmp == CSTART || ctmp == CSTOP)
        continue ;
}
/* end XON/XOFF output flow control section */

/* stash character in r_buf */

```

```
if ( tp->t_rbuf.c_ptr == NULL )    /* why not check for c_count == 0 */
    continue ;

if (tp->t_cflag & PARENB) /* strip off parity bit(s) if parity enabled */
{
    switch (tp->t_cflag & CSIZE) /* number of bits in character */
    {
        case CS7:
            c &= 0x7f;
            break;
        case CS6:
            c &= 0x3f;
            break;
        case CS5:
            c &= 0x1f;
            break;
    }
}

if (tp->t_iflag & ISTRIP)
    c = c & 0177;

*tp->t_rbuf.c_ptr++ = c ;
--tp->t_rbuf.c_count ;

if ( c == 0377 && tp->t_iflag&PARMRK ) /* 0377 is 0377 0377 */
{
    /* this avoids ambiguity */
    *tp->t_rbuf.c_ptr++ = 0377 ; /* when parity errors are marked */
    --tp->t_rbuf.c_count ;      /* and passed to the LDR */
}

/* adjust c_ptr correctly */
tp->t_rbuf.c_ptr -= tp->t_rbuf.c_size - tp->t_rbuf.c_count;
(*linesw[tp->t_line].l_input)(tp) ; /* LDR input */
}
break;
```

LDR TO DRIVER COMMUNICATION

The LDR, not the driver, contains the information to determine when input flow should be stopped or started. The LDR calls the device driver which implements the input flow control. There are a number of other times when the LDR needs to communicate with the driver. The driver *proc* routine is used for LDR to driver communication. The entry points in the *proc* routine that must be provided are:

Starting output

- Suspending output
- Resuming output
- Breaking transmission
- Transmitting buffer flush
- Receiving buffer flush
- Logging an overrun error

The format for the *proc* routine is:

```
driverproc(tp, command);  
struct tty *tp;      /* tty structure */  
int command;         /* command to be performed */
```

The name of the routine for the NEC driver is *necproc()*.

These commands are handled by the *proc* routine:

T_OUTPUT

Start output to the device. Call the LDR output routine to get a *t_tbuf* if the current buffer is empty.

T_BLOCK

Stop the flow of input

T_UNBLOCK

Start the flow of input

T_SUSPEND

Stop the flow of output

T_RESUME

Start the flow of output

T_BREAK

Send a break stream to the device for 1/4 second. Normally this is done by setting a hardware status register to send the break and scheduling a timeout in 1/4 second. After the 1/4 second elapses, call *necproc(tp, T_TIME)*.

T_TIME

Clear the break condition on the device. Start output to the device.

T_WFLUSH

Flush the current transmit buffer and start the flow of output starting with the next *t_tbuf*.

T_RFLUSH

If blocked, unblock. If unblocked, return.

T_LOG_FLUSH

Raw input queue overflow has occurred; empty read queue and UNBLOCK input.

NECPROC LISTING

```

/*-----
 * pnec_proc.c -- focal point communication between LDR and the driver
 *-----
 */
necproc(tp, cmd)
    register struct tty *tp;      /* tty pointer */
    {
        register c;
        register struct device *rp; /* device register pointer */
        int priority;              /* priority incoming */
        int dev;                   /* minor number a=0 b=1 */
        extern ttrstrt();

        dev = tp - nec_tty;        /* minor is 0 or 1 */
        rp = nec_addr + dev;       /* register pointer set accordingly */
        switch (cmd) {

            case T_TIME:
                # include "pnec.time.c" /* timeout expiration handler */

            case T_WFLUSH:           /* flush transmit buffer note no break */
                tp->t_tbuf.c_size -= tp->t_tbuf.c_count;
                tp->t_tbuf.c_count = 0;

            case T_RESUME:           /* resume output, note no break */
                tp->t_state &= TTSTOP; /* clear suspend flag */

            case T_OUTPUT:          /* restart output */
                start:               /* where to go to start transmitting */
                # include "pnec.output.c" /* output start handler */

            case T_SUSPEND:         /* suspending output is just */
                tp->t_state != TTSTOP; /* setting this flag */
                break;

            case T_BLOCK:

```

```

#   include "pnec.block.c"           /* block handler */

    case T_RFLUSH:                    /* read flush */
        if (!(tp->t_state&TBLOCK)) /* if blocked unblock */
            break;

    case T_UNBLOCK:
#   include "pnec.unblock.c"         /* unblock handler */

    case T_BREAK:                     /* send a break for 1/4 second */
#   include "pnec.break.c"           /* break handler */

    case T_LOG_FLUSH:
        check_tally( dev, T_OVERFLOW );
        break ;
    }

```

DRIVERPROC(TP, T_OUTPUT)

This routine is called from the LDR to start output. If the device is not busy, the routine transmits a character to start receiver interrupts. If the device is busy, a character can not be sent until the next receiver ISR. Busy conditions are:

BUSY

The device is in the process of transmitting a character.

TTSTOP

The driver is in a output flow control hold state; output is suspended.

TIMEOUT

A certain amount of time must pass before giving the device the next character.

If there are no characters in the *t_tbuf* to transmit, the LDR *l_output* routine is called to fill the *t_tbuf*.

NOTE: This routine is also called from the driver when the transmitter ISR routine finds that *t_tbuf* is empty.

PNEC.OUTPUT.C LISTING

```

/*-----
* pnec.output.c -- T_OUTPUT routine. Called to start the output
* from the LDR. Also called from the driver when transmit isr has
* no characters to transmit.
* If the transmission of characters is impossible then this routine
* did not have to be called so return immediately. The reasons
* for not being able to transmit characters are:
*   BUSY - character is being output
*   TTSTOP - output has been suspended
*   TIMEOUT - output has been stopped for a certain time period
*-----
*/
{
register struct ccblock *tbuf;          /* transmit buffer */

priority = spl3();
if( tp->t_state & (TTSTOP|BUSY|TIMEOUT) )
{
/* impossible to transmit chars */
splx( priority );
break;
}

tbuf = &tp->t_tbuf;                    /* set tbuf to the t_buf */
if( tbuf->c_ptr == NULL || tbuf->c_count == 0 )
{
/* need a new t_buf */
if( tbuf->c_ptr )
/* if tbuf has been initialized */
tbuf->c_ptr -= tbuf->c_size; /* reset the buffer pointer */
/* get another t_buf */
if( !(CPRES & (*linesw[tp->t_line].l_output)(tp)) )
{
/* if no chars in t_buf */
splx( priority );
/* there is nothing to transmit */
break;
/* so quit */
}
}

if( tbuf->c_count )
{
/* if there are chars to transmit */
/* take one out of tbuf and */
/* transmit it */
tp->dbuf = *tbuf->c_ptr++;
tp->t_state != BUSY;
}
splx( priority );
break;
}

```


DRIVERPROC(TP, T_BLOCK)

The functions this routine performs are:

- Block further input; the high water mark has been reached and no further input should be permitted.
- Mark the driver state to show that input is blocked; primarily used by the LDR.
- If the ready/busy input flow control protocol is being used, stop input by turning off RTS.
- If the XON/XOFF input flow control protocol is being used, transmit an XOFF character. If the device is busy, inform the transmit ISR to transmit an XOFF character.

NECPROC(TP, T_BLOCK) LISTING

```

/*-----
 * pnec.block.c -- block the input of characters.
 * XON/XOFF or ready/busy protocol is allowed
 * XON/XOFF - send a XOFF character or tell the xmit isr to send one.
 * ready/busy - turn off RTS
 *-----
 */
tp->t_state != TBLOCK;          /* Let the xmit isr know */
tp->t_state &= TTXON;           /* clear the send XON flag */

if( tp->t_iflag & IRTS )        /* ready/busy stuff */
{
    priority = spl3();
    rp->csr = RP5;              /* turn off RTS */
    rp->csr = CRS[dev].cr5 = CRS[dev].cr5 & ~RTS;
    splx( priority );
}

if( tp->t_iflag & IXOFF )       /* disable input using XON/XOFF protocol */
{
    priority = spl4();
    if( tp->t_state & BUSY )     /* cant send an XOFF now, so */
        tp->t_state != TTXOFF; /* let the xmit interrupt send it */
    else
    {
        /* send the XOFF now */
        tp->t_state != BUSY;
        rp->dbuf = nec_xoff;
    }
}

```

```
        }
        splx( priority ) ;
    }
    break;
```

DRIVERPROC(TP, T_UNBLOCK)

The low water mark has been reached so input is permitted. These functions are performed:

- Reset the block flag.
- If ready/busy input flow control protocol is being used, turn on RTS.
- If XON/XOFF input flow control protocol is being used, transmit an XON character or inform the transmit ISR to transmit an XON character.

PNEC.UNBLOCK.C LISTING

```
/*-----
 * pnec.unblock.c -- allow the input of characters.
 * XON/XOFF or ready/busy protocol is allowed
 * XON/XOFF - send a XON character or tell the xmit isr to send one.
 * ready/busy - turn on RTS
 *-----
 */
tp->t_state &= (TTXOFF|TBLOCK); /* clear block state and no XOFF to send */

if( tp->t_iflag & IRTS )      /* ready busy stuff */
{
    priority = spl3() ;
    rp->csr = RP5 ;          /* turn on RTS */
    rp->csr = CRS[dev].cr5 = CRS[dev].cr5 | RTS ;
    splx( priority ) ;
}

if( tp->t_iflag & IXOFF )    /* XON/XOFF stuff */
{
    priority = spl4() ;
    if( tp->t_state & BUSY )
        tp->t_state != TTXON; /* send an XON during xmit interrupt */
    else
    {
```

```

        tp->t_state != BUSY ; /* send an XON now */
        rp->dbuf = nec_xon;
    }
    splx( priority ) ;
}
break ;

```

DRIVERPROC(TP, T_SUSPEND-TP, T_RESUME)

These routines are called to suspend and resume output. Their implementation is described in the main line of *necproc*.

NOTE: The flag TTSTOP identifies this state.

DRIVERPROC(TP, T_BREAK)

This routine is called from the upper layers when a break is to be sent to the terminal for 1/4 second. Functions performed are:

- Set hardware to send break.
- Set the state to TIMEOUT.
- Schedule a wakeup in 1/4 of a second to reset the hardware.

PNEC.BREAK.C LISTING

```

/*-----
 * pnec.break.c -- T_BREAK routine. Called when a break is to
 * be sent to a terminal.
 * A break is sent for 1/4 of a second or HZ/4 ticks.
 * The routine ttrstrt will make the call necproc(tp, T_TIME)
 *-----
 */
priority = spl3() ; /* disable nec interrupts */
rp->csr = RP5 ; /* this is a hardware function */
rp->csr = CRS[dev].cr5 |= SEND_BREAK ;
splx( priority ) ; /* allow interrupts */

```



```

tp->t_state != TIMEOUT ;           /* timeout for 1/4 second */
timeout( ttrstrt, tp, HZ/4 );      /* then execute ttrstrt */
break ;

```

DRIVERPROC(TP, T_TIME)

A timeout has expired because the 1/4 second break transmission has completed. These functions are performed:

- Clear the timeout flag in the state variable.
- Reset the hardware to stop the transmission of the break.
- Start the transmission of characters

PNEC.TIME.C LISTING

```

/*-----
 * pnec.time.c -- T_TIME routine. Called when a timeout for
 * a terminal has expired.
 * The break will be turned off whether it was on or not.
 * control is transferred to start (see T_OUTPUT) to start the
 * transmission of characters.
 *-----
 */
tp->t_state &= TIMEOUT;           /* clear the timeout state */
switch( tp->t_cflag & CSIZE ) { /* reset break HW flag */
case CS5:                         /* this stuff is calculated because */
    c = XCBITS5 ;                 /* the same HW reg controls transmit bits */
    break ;                       /* and break sending */
case CS6:
    c = XCBITS6 ;
    break ;
case CS7:
    c = XCBITS7 ;
    break ;
case CS8:
    c = XCBITS8 ;
    break ;
}
priority = spl3() ;               /* disable interrupts */
rp->csr = RP5 ;                   /* Don't turn off anything but break */
rp->csr = DTR | TRANS_ENB | RTS | c ;
splx( priority ) ;               /* enable interrupts */

goto start;                       /* start the transmission of characters */

```

PROC COMMAND - FLAG AFFECTED

Figure 15 is a summary of the process commands and their influence on the related function.

Command	t_state	Description
T_BLOCK	T_BLOCK !TTXON TTXOFF	block input
T_UNBLOCK	!T_BLOCK !TTXOFF TTXON	start input
T_SUSPEND	TTSTOP	stop output
T_RESUME	!TTSTOP	start output
T_OUTPUT	none	get t_buf, output char(s)
T_BREAK	TIMEOUT	start break
T_TIME	!TIMEOUT	stop break, output char(s)
T_WFLUSH	!TTSTOP	reset t_buf, resume
T_RFLUSH	!TBLOCK !TTXOFF TTXON	only if blocked
T_LOG_FLUSH	none	input way to fast
T_INPUT	none	none
T_PARM	none	none
T_SWTCH	none	none

Figure 15. Process Commands

CLOSE ROUTINE

The *close* routine is used to call the LDR's *l_close* routine. If the DTR is to be turned off at close, the hardware must be informed.

PNEC.CLOSE.C LISTING

```
/*-----  
* pnec.close.c -- close routine  
*-----  
*/  
necclose(minor_d, flag)  
dev_t minor_d;          /* minor device number */  
int flag;                /* not used */  
{  
    register struct tty *tp;      /* pointer to tty for minor device */  
    register struct device *rp;   /* register of device to close */  
    int priority;                /* original priority */  
  
    necsave ("necclose");  
  
    tp = &nec_tty[minor_d];       /* tty pointer */  
    rp = nec_addr + minor_d;      /* register pointer */  
  
    (*linesw[tp->t_line].l_close)(tp); /* LDR close */  
    if ( minor_d = minor(consdev)) /* console is now closed */  
        console_open = FALSE ;  
  
    priority = spl3() ;           /* critical region */  
    rp->csr = RP3 ;  
    rp->csr = RCBITS8 ! AUTO_ENB; /* disable the receiver */  
    if ( tp->t_cflag & HUPCL && !(tp->t_state&ISOPEN) )  
    {                             /* hang up on close */  
        rp->csr = RP5 ;           /* turn off DTR */  
        if (tp->t_cflag & PARENB)  
            rp->csr = RTS ! XCBITS7 ! TRANS_ENB ;  
        else  
            rp->csr = RTS ! XCBITS8 ! TRANS_ENB ;  
    }  
    splx( priority ) ;           /* end critical region */  
}
```


IOCTL ROUTINE

The *ioctl* routine calls the LDR *l_ioctl* procedure. The return value of the *l_ioctl* procedure tells whether the hardware configuration has been changed. If the hardware needs to be configured, then start the configuration from the beginning, using the bit flags in *t_cflag*. The hardware configuration is hardware dependent.

ERROR HANDLING

When errors are detected, the input bit flags *t_iflag* explain how errors are to be handled.

INPCK

Input parity check. If set, these types of error are handled:

- Parity errors.
- Framing errors that are not break characters.
- Overrun errors.

IGNBRK

Ignore break. A break causes a framing error. If a framing error occurs and the character read is 0377 (all binary 1's), the driver should ignore the condition if this flag is set. If the flag isn't set, this code should be executed:

```
signal(tp->t_pgrp, SIGINT);  
ttyflush(tp, (FREAD|FWRITE));
```

PARMRK

Parity framing and overrun errors should be marked with the three character sequence

0377 0 x

NOTE: x is the character found to have a parity error.

IGNPAR

Ignore input errors.

Virtual Disk Driver

Chapter 10.

A Virtual Disk Driver

INTRODUCTION TO DISK DRIVERS

Disk device drivers control devices that permit random access of fixed size blocks. Blocks on the disk are numbered from zero to the size of the device minus one. The blocks are cached in the kernel for eventual reuse. Two interfaces are generally provided by a disk driver. A block device interface accessed through the switch table *bdevsw* is the most frequently used interface. A character device interface accessed through the switch table *cdevsw* and is used for special purpose access.

Block handling routines

The Device Independent Kernel is a layer above the device driver (Figure 1).

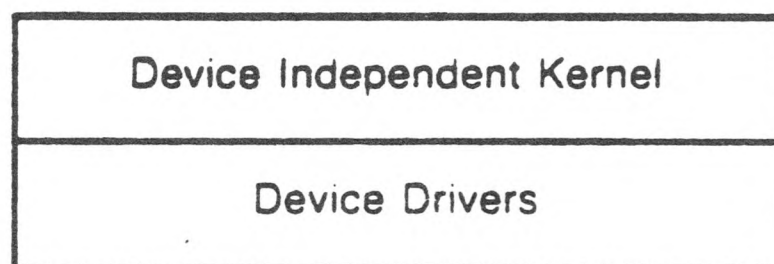


Figure 1. Device Independent Layer

The kernel has a layer of software, called the Block Handling routines, to increase the efficiency of handling disk devices. This modular design simplifies the integration of new disk drivers. This layer keeps a cache list of disk blocks in memory. Blocks to be read or written to disk pass through this cache. Commonly used blocks remain in the cache and are therefore accessed without the driver being invoked to physically retrieve the blocks. Caching is advantageous since the disk device is slow compared to processor speeds. The driver is concerned with efficient retrieval of disk blocks, not caching.

Figure 2 shows the relationship between the device independent kernel, the block handling routines, and the device driver.

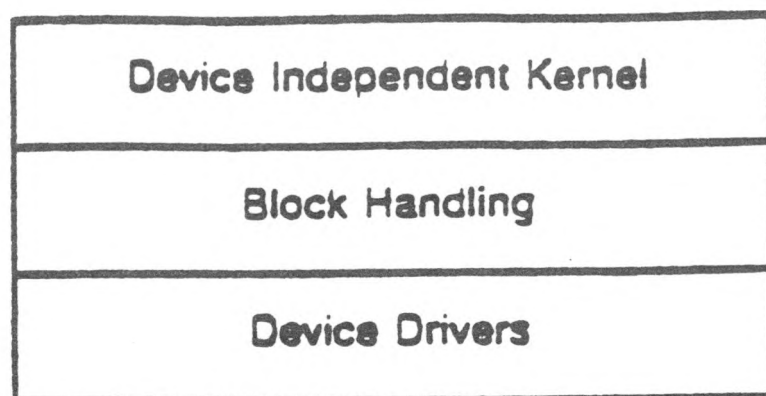


Figure 2. Block Device Driver Layers

Buffers and Buffer Headers

A reserved area of kernel memory is used for buffers to hold data to be written to the device or data that has been read from the device. Each kernel buffer is 1024 bytes long.

When a process wants to copy a data structure to disk, it is actually copied to a kernel buffer at processor speeds and not immediately transferred to disk. When a block is requested, the system buffers (cache) are searched to see if the block is already in memory. If so, the system uses the block found in memory. If not, the block is read into a system buffer. This reduces I/O traffic since frequently used blocks stay in memory.

This buffering also permits a process to handle any number of bytes at a time; the system buffers hold any unwanted data. For example, if a process writes ten bytes to a file on disk, the block that contains the ten bytes is read from disk into a kernel buffer. The ten bytes are then replaced with the user's data. The kernel buffer is then put on the list of kernel buffers that needs to be written to disk at a later time.

Each kernel buffer has a buffer header associated with it. The buffer header describes the transaction. Fields like device number, block number, kernel buffer pointer, etc. are contained in the buffer header. The buffer header also describes an operation that needs to be performed by a disk driver. The number of buffers and buffer headers is determined by the *buffer's* configuration parameter.

Buffer Header Description

The definition of a buffer header is found in the file */usr/include/sys/buf.h*. Here is the buffer header structure definition from the file.


```

struct buf
{
    int b_flags;          /* see defines below */
    struct buf *b_forw;   /* headed by d_tab of conf.c */
    struct buf *b_back;   /* " */
    struct buf *av_forw;  /* position on free list, */
    struct buf *av_back;  /* if not BUSY*/
    dev_t b_dev;          /* major+minor device name */
    int b_umd;            /* unibus map descriptor */
    unsigned b_bcount;    /* transfer count */
    union {
        caddr_t b_addr;   /* low order core address */
        int *b_words;     /* words for clearing */
        struct filsys *b_filsys; /* superblocks */
        struct dinode *b_dino; /* ilist */
        daddr_t *b_daddr;  /* indirect block */
    } b_un;
};

#define paddr(X) (paddr_t)(X->b_un.b_addr)

    daddr_t b_blkno;      /* block # on device */
    char b_error;         /* returned after I/O */
    unsigned int b_resid;  /* words not transferred after error */
    time_t b_start;       /* request start time */
    struct proc *b_proc;   /* process doing physical or swap I/O */
    struct buf *b_drsrbp;  /* drs_stuff - bp of owning box */
};

/*
 * These flags are kept in b_flags.
 */
#define B_WRITE 0 /* non-read pseudo-flag */
#define B_READ 01 /* read when I/O occurs */
#define B_DONE 02 /* transaction finished */
#define B_ERROR 04 /* transaction aborted */
#define B_BUSY 010 /* not on av_forw/back list */
#define B_PHYS 020 /* Physical I/O potentially using UNIBUS map */
#define B_MAP 040 /* This block has the UNIBUS map allocated */
#define B_WANTED 0100 /* issue wakeup when BUSY goes off */
#define B_AGE 0200 /* delayed write for correct aging */
#define B_ASYNC 0400 /* don't wait for I/O completion */
#define B_DELWRI 01000 /* don't write till block leaves available list */
#define B_OPEN 02000 /* open routine called */
#define B_STALE 04000

#define B_DRSEBITS 0x0f000000 /* bits used by drs_code */
#define B_MASTER 0x08000000 /* B_BUSY in home box drs_code */
#define DRS_BP_BITS 24 /* # of bits to sift to put in node # drs_code */

```

The fields in the buffer header that relate to driver operation are:

b_flags

Bit flags

B_ERROR

Error during I/O operation

B_READ

If set, the operation is a read; otherwise a write.

B_PHYS

The operation is a direct transfer from user space to disk without passing through the block handling routines.

av_forw

Forward pointer on the device driver I/O queue.

av_back

Backward pointer if device driver I/O queue is doubly linked.

b_dev

Major and minor device number of the device the block is found on.

b_bcount

Number of bytes to be transferred. This is typically 1024 when the block handling routines are used. When the character device interface is used, this value could be any number. Normally only integral number of blocks are actually transferred.

b_un

Address of primary memory buffer to be transferred. When using the block handling routines, this is always a kernel buffer.

b_blkno

Block number on disk. The operating system assumes that all physical blocks are 512 bytes in length.

b_error

If the bit B_ERROR is set in *b_flags*, this error number is copied to the process's copy of the variable *errno*.

b_resid

The number of bytes in the request that were not transferred. If, for example, 2048 bytes were requested and only 512 bytes were transferred, then the value of *b_resid* should be set to 1536.

b_proc

Pointer to process table entry of the physical I/O process that requested the block.

The fields in the *buf* structure are used by the block handling routines and can not be altered indiscriminately by the driver. Alterable fields of the *buf* structure are:

b_flags

The flag *B_ERROR* can be set.

av_forw

In complete control of the driver.

av_back

In complete control of the driver.

b_error

Error number returned to user program in variable *errno*.

b_resid

Number of bytes not transferred.

Driver I/O Queue

A driver I/O queue is an I/O request queue maintained by the driver. Buffer headers--associated with buffers waiting for the data to be read into them, or buffers waiting for the data to be written back to disk--are linked onto this queue. The *iobuf* structure is an anchor for this queue and contains two fields used for list manipulation:

b_actf

Pointer to the beginning of the driver I/O queue

b_actl

Pointer to the end of the driver I/O queue

Figure 3 provides a representation of a doubly linked driver I/O queue.

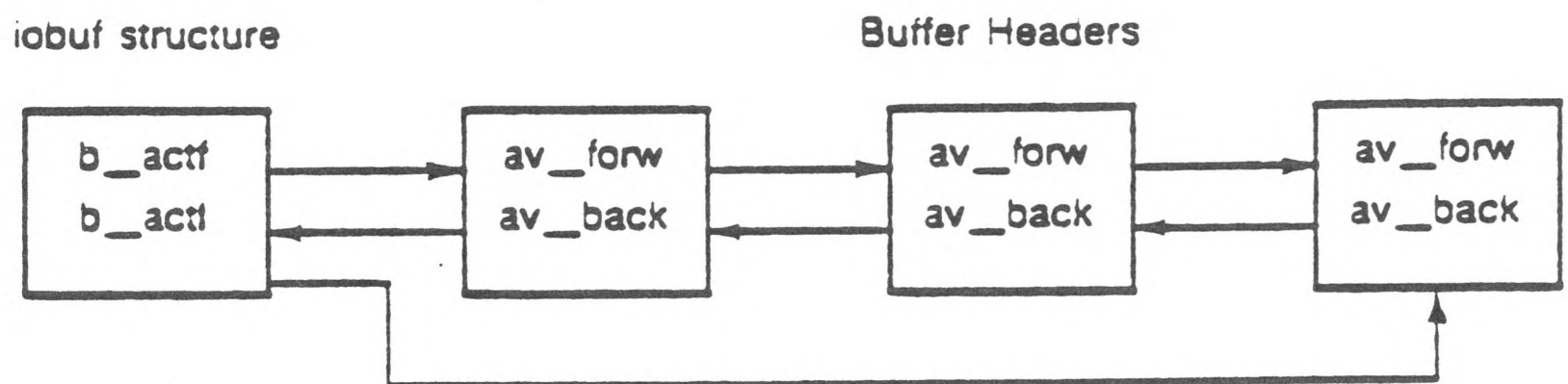


Figure 3. Driver I/O Queue

The driver I/O queue is only used by the driver, so a singly linked list could be used. Typically, there is a driver I/O queue for each device controlled by the driver.

Block Device Interface

The block device interface is used for communication between the block handling routines and the device driver. The device driver is responsible for 512-byte physical blocks of data. The block handling routines generally request these physical blocks two at a time. Most disk devices can be requested to transfer both of the physical blocks in one operation. During initialization, when the boot block and super block are

read, only one physical block is requested.

Bdevsw

The block device driver is called using the *bdevsw* table. The similarities between *cdevsw* and *bdevsw* are apparent. The driver is identified by its major device number. *Bdevsw* is described in */usr/include/sys/conf.h*:

```
/*
 * Declaration of block device switch. Each entry (row) is
 * the only link between the main unix code and the driver.
 * The initialization of the device switches is in the file conf.c.
 */
struct bdevsw {
    int (*d_open)();
    int (*d_close)();
    int (*d_strategy)();
    int (*d_print)();
};
extern struct bdevsw bdevsw[];
```

There are a few differences between the *bdevsw* and *cdevsw* tables. There is no *d_ioctl* routine in the *bdevsw* table. The routine *d_strategy* replaces the *d_read* and the *d_write* routines. The *d_print* routine is maintained for historical compatibility and is generally not used.

VIRTUAL DISK DRIVER DESCRIPTION

The virtual disk driver (*vd*) allocates an array of 512 byte blocks that can be used to store data. The following C statements describe the driver's disk storage:

```
#define VDPBLK 0x200          /* physical block size in bytes */
#define VDPBLKS 0x100        /* number of physical blocks on disk */
struct vdblock
{
    char data [VDPBLK];
```

```

    }
    struct vdblock vddisk[VDPBLKS];

```

The address of block *b* in the disk is described with this C statement:

```

char * block_addr;
block_addr = &vddisk[b];

```

Vdopen Definition

The *vdopen* routine is called using the *fibdevswfR* table. The minor device is validated. For a flex disk, the read/write notch on the flex could be sensed. On a hard disk, the existence of the disk could be verified.

Vdopen listing

```

/*-----
 * vdopen - open call common for block and character device interface
 *-----
 */
vdopen(minor_d, flags)
dev_t  minor_d;      /* minor device number */
int     flags;        /* flags are ignored */
{
    print1("vdopen0");
    if (minor_d >= vd_cnt)      /* minor device does not exist */
        u.u_error = ENXIO;
}

```

Virtual Driver Strategy Definition

The strategy routine performs both read and writes from the disk. A buffer header that fully describes the operation is passed as a parameter. When

controlling a "real" device, the buffer header is put on the driver I/O queue, after which control is passed back to the block handling routines. It's up to the ISR to signal completion of the operation by calling *iodone(k)*. *Vd* has no ISR, so the operation is completed immediately and *iodone* is called. The buffer header is first verified to be correct. If the requested block is one block beyond the end of the device, an end-of-file condition exists which is not considered an error. If other blocks beyond the last block have been requested, an error condition exists.

The routine *blt(k)* is a fast way to copy buffers. *Bl*t doesn't do error checking so it should not be used for copying data into and out of user space. *Iodone(k)* is called to signal the upper layers that the operation described by the buffer header has been completed.

Vdstrategy listing

```
/*-----
 * vdstrategy - strategy routine
 *-----
 */
vdstrategy(bp)
struct buf *bp;
{
    int physblk;

    if (vdverify(bp))
        vdrequest(bp);
    iodone(bp);
}

/*-----
 * vdverify - verify buffer pointer, return true to continue with the
 * request or false to stop processing
 *-----
 */
vdverify (bp)
struct buf * bp;
{
    if (bp->b_blkno == VDPBLKS)    /* end of file, not really an error */
    {
```



```

        bp->b_resid = bp->b_bcount;    /* number of bytes not yet transfered */
        return FALSE;
    }
    else if (bp->b_blkno > VDPBLKS)    /* error: block number is to hi */
    {
        print1("error block number to high0);
        bp->b_resid = bp->b_bcount;    /* number of bytes not yet transfered */
        bp->b_flags |= B_ERROR;
        return FALSE;
    }
    else
        return TRUE;
}

/*-----
 * vrequest - fullfill a request
 *-----
 */
vrequest(bp)
struct buf * bp;
{
    print2("buffered %x0, bp->b_blkno);
    if (bp->b_flags & B_READ) /* read or write operation */
        b1t (bp->b_un.b_addr, &vddisk[bp->b_blkno], bp->b_bcount);
    else
        b1t (&vddisk[bp->b_blkno], bp->b_un.b_addr, bp->b_bcount);
}

```

Virtual Disk Driver Complete listing

```

/*-----
 * vd.c -- virtual disk driver
 *-----
 */

/*-----
 * include files
 *-----
 */
#include <sys/sysmacros.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/signal.h>
#include <sys/errno.h>
#include <sys/dir.h>

```

Chapter 10

```
#include <sys/file.h>
#include <sys/user.h>
#include <sys/buf.h>
#include <sys/iobuf.h>
#include <sys/b0.h>
#include <sys/proc.h>

/*-----
 * global definitions
 *-----
 */
#define VDPBLK      0x200      /* physical block size in bytes */
#define VDPBLKMSB  0xfffffe00 /* physical block most sig bit mask */
#define VDPBLKLSB  0x1ff      /* physical block least sig bit mask */
#define VDPBLKSHIFT 0x09      /* power of 2 equal to VDPBLK */
#define VDPBLKS    0x100      /* number of physical blocks on disk */

struct vdblock
{
    char    data[VDPBLK];
};
struct vdblock vddisk[VDPBLKS];

struct buf vd_buf;          /* internal buffer */

/*-----
 * debugging routines
 *-----
 */
#define DEBUG
#ifdef DEBUG
    int vd_debug = 1;
#   define print1(a) vd_printf(a)
#   define print2(a,b) vd_printf(a,b)
#   define print3(a,b,c) vd_printf(a,b,c)
#   define print4(a,b,c,d) vd_printf(a,b,c,d)
#   define print5(a,b,c,d,e) vd_printf(a,b,c,d,e)
    vd_printf(a,b,c,d,e)
    {
        if (vd_debug) printf(a,b,c,d,e);
    }
#else
#   define print1(a)
#   define print2(a,b)
#   define print3(a,b,c)
#   define print4(a,b,c,d)
#   define print5(a,b,c,d,e)
#endif
/*-----
```

```

* external references
*-----
*/
extern int vd_cnt;          /* number of minor devices */

#define TRUE 1
#define FALSE 0

/*-----
* vdopen - open call common for block and character device interface
*-----
*/
vdopen(minor_d, flags)
dev_t  minor_d;      /* minor device number */
int    flags;        /* flags are ignored */

{
    print1("vdopen0");
    if (minor_d >= vd_cnt)    /* minor device does not exist */
        u.u_error = ENXIO;
}

/*-----
* vdstrategy - strategy routine
*-----
*/
vdstrategy(bp)
struct buf *bp;
{
    int physblk;

    if (vdverify(bp))
        vdrequest(bp);
    iodone(bp);
}

/*-----
* vdverify - verify buffer pointer, return true to continue with the
* request or false to stop processing
*-----
*/
vdverify (bp)
struct buf * bp;
{
    if (bp->b_blkno == VDPBLKS) /* end of file, not really an error */
    {
        bp->b_resid = bp->b_bcount; /* number of bytes not yet transfered */
        return FALSE;
    }
    else if (bp->b_blkno > VDPBLKS) /* error: block number is to hi */

```

Chapter 10

```
    {
        print1("error block number to high0);
        bp->b_resid = bp->b_bcount; /* number of bytes not yet transfered */
        bp->b_flags |= B_ERROR;
        return FALSE;
    }
else
    return TRUE;
}

/*-----
 * vrequest - fullfill a request
 *-----
 */
vrequest(bp)
struct buf * bp;
{
    print2("buffered %x0, bp->b_blkno);
    if (bp->b_flags & B_READ) /* read or write operation */
        blt (bp->b_un.b_addr, &vddisk[bp->b_blkno], bp->b_bcount);
    else
        blt (&vddisk[bp->b_blkno], bp->b_un.b_addr, bp->b_bcount);
}

/*-----
 * vdclose - nothing
 *-----
 */
vdclose()
{
    print1("vdclose: not implemented0);
}

/*-----
 * vdprint - needed to get rid of undefined references
 *-----
 */
vdprint()
{
    print1("vdprint: not implemented0);
}

/*-----
 * vdioc1 - not implemented
 *-----
 */
vdioc1()
{
    print1("vdioc1: not implemented0);
}
```


CHARACTER DEVICE PORTION OF THE DRIVER

A character device interface is also provided by disk drivers. This interface is defined in exactly the same way as any other character device. The character device portion for a disk driver has the following unique characteristics.

- An *ioctl* call is provided as a way to format the disk, define disk characteristics like number of sectors read, and to set the bad block table, etc.
- A read and write system call requires that the driver transfer data directly from user buffers to disk blocks without using the kernel buffers. A special kernel call, *physio(k)*, helps perform this task.

This interface is sometimes called the "physical" device interface. From the process's point of view, data is moved directly between a physical device and process memory (no kernel buffers). The read and the write routines aren't important and are described in the following example.

Character Read and Write Listing

```
struct buf vd_buf;
/*-----
 * vread - character device interface read routine
 *-----
 */
vread(minor_d)
dev_t minor_d;
{
    physio(vdstrategy, &vd_buf, minor_d, B_READ);
}
```

```
/*-----  
 * vdwrite - character device interface write routine  
 *-----  
 */  
vdwrite(minor_d)  
dev_t minor_d;  
{  
    physio(vdstrategy, &vd_buf, minor_d, B_WRITE);  
}
```

PHYSIO I/O

The *physio* call is:

```
physio(strategy, bp, minor_d, flag)  
int (strategy *)();          /* drivers strategy routine */  
struct buf * bp;             /* buffer header */  
dev_t minro_d;               /* minor device number */  
int flag;                    /* 0 - Write 1 - Read */
```

The buffer pointer provided is filled with values from the user structure and the parameters. The driver's strategy routine is then called to perform the operation. Paging systems break one large operation into 64K pieces and call the driver's strategy routine. Swapping systems call the strategy routine only once. The portion of the process required for the I/O operation is also mapped and locked into memory by *physio*.

Physical I/O Strategy Definition

The strategy routine for the virtual disk driver works correctly with no changes. The *blt* call, however, does no error checking and could cause disastrous results. It has therefore been replaced by *copyin(k)* and *Icopyout(k)* that do error checking.

Physical I/O Strategy Listing

```
/*-----
 * vdstrategy - strategy routine
 *-----
 */
vdstrategy(bp)
struct buf *bp;
{
    int physblk;

    if (vdverify(bp))
        vdrequest(bp);
    iodone(bp);
}

/*-----
 * vdverify - verify buffer pointer, return true to continue with the
 * request or false to stop processing
 *-----
 */
vdverify (bp)
struct buf * bp;
{
    if (bp->b_blkno == VDPBLKS)                /* end of file, not really an error */
    {
        bp->b_resid = bp->b_bcount;            /* number of bytes not yet transfered */
        return FALSE;
    }
    else if (bp->b_blkno > VDPBLKS)             /* error: block number is to hi */
    {
        print1("error block number to high0);
        bp->b_resid = bp->b_bcount;            /* number of bytes not yet transfered */
        bp->b_flags |= B_ERROR;
        return FALSE;
    }
    else
        return TRUE;
}

/*-----
 * vdrequest - fullfill a request
 *-----
 */
vdrequest(bp)
struct buf * bp;
{
    if (bp->b_flags & B_PHYS)                  /* physio call */
    {
```

```
    print2("physio %x0, bp->b_blkno);
    if (bp->b_flags & B_READ)          /* read or write operation */
        copyout (&vddisk[bp->b_blkno], bp->b_un.b_addr, bp->b_bcount);
    else
        copyin (bp->b_un.b_addr, &vddisk[bp->b_blkno], bp->b_bcount);
    }
else                                  /* normal buffered io call */
{
    print2("buffered %x0, bp->b_blkno);
    if (bp->b_flags & B_READ)          /* read or write operation */
        blt (bp->b_un.b_addr, &vddisk[bp->b_blkno], bp->b_bcount);
    else
        blt (&vddisk[bp->b_blkno], bp->b_un.b_addr, bp->b_bcount);
    }
}
```

MEMORY MANAGEMENT

Disk controllers generally use DMA to transfer data from disk to memory. Memory accesses by the CPU and by DMA devices use the two level memory management unit to access primary memory.

The function of the memory management unit is to take logical addresses provided by the MC68010 or MC68020 CPU and convert those addresses into physical addresses of memory bytes. Figure 4 provides a simplified diagram of memory management.

The MMU can best be understood by studying Figure 5 while reading the following description. The MMU supports a number of contexts simultaneously. Each context describes the mapping currently being used by a process.

Having more than one context permits many processes to be loaded into the MMU and context switching to be done by simply changing the value in the context register. Context 31 is reserved for use by DMA devices. All DMA memory access automatically forces context 31 for the duration of the transfer. This is enforced by the hardware and cannot be changed.

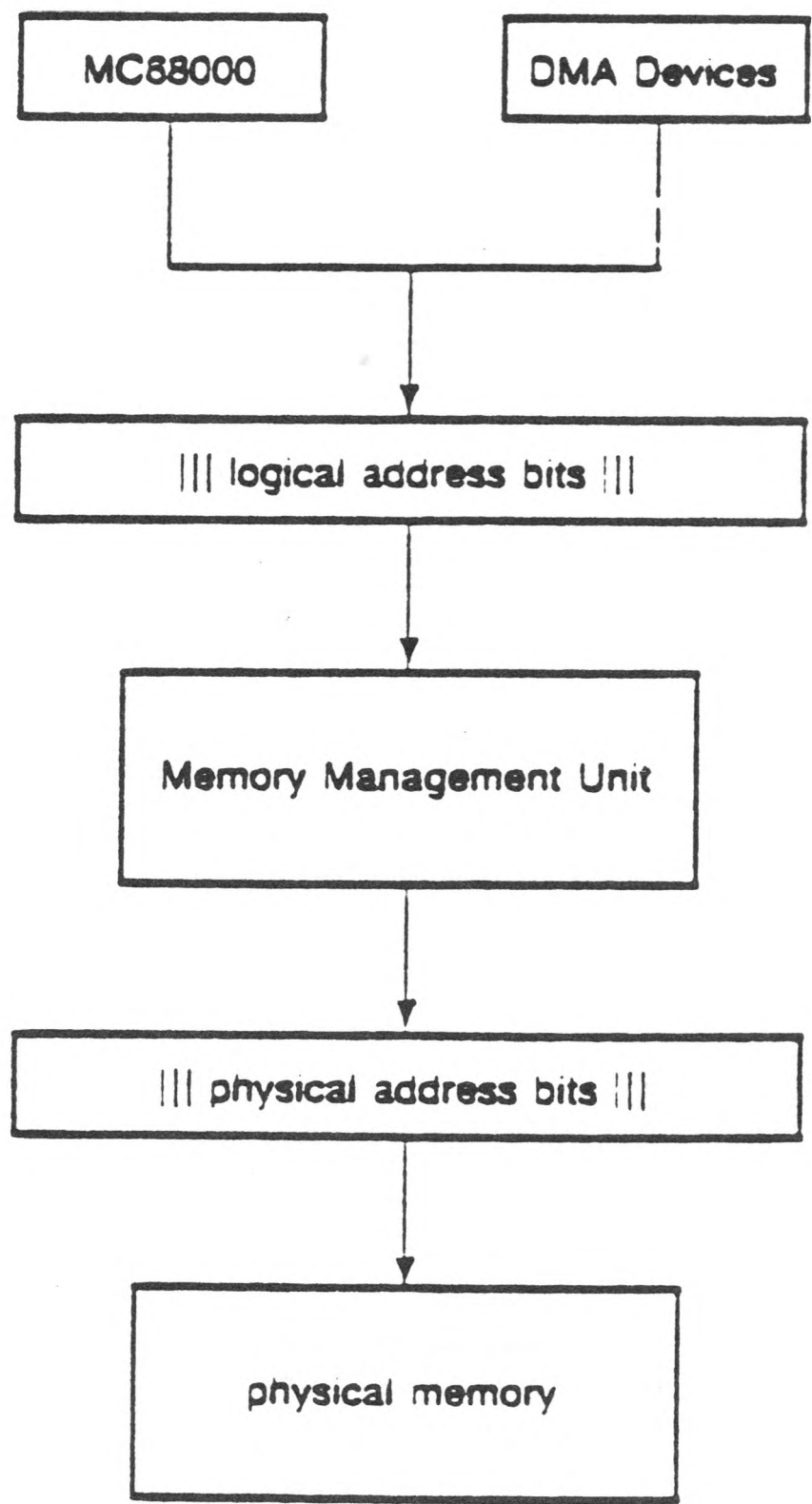


Figure 4. Memory Management Overview

Each context consists of a number of segments. The context register, along with the most significant bits in a logical address, identify one cell in the segment map. Each cell in the segment map contains an index into the page map. This index is called the page block number.

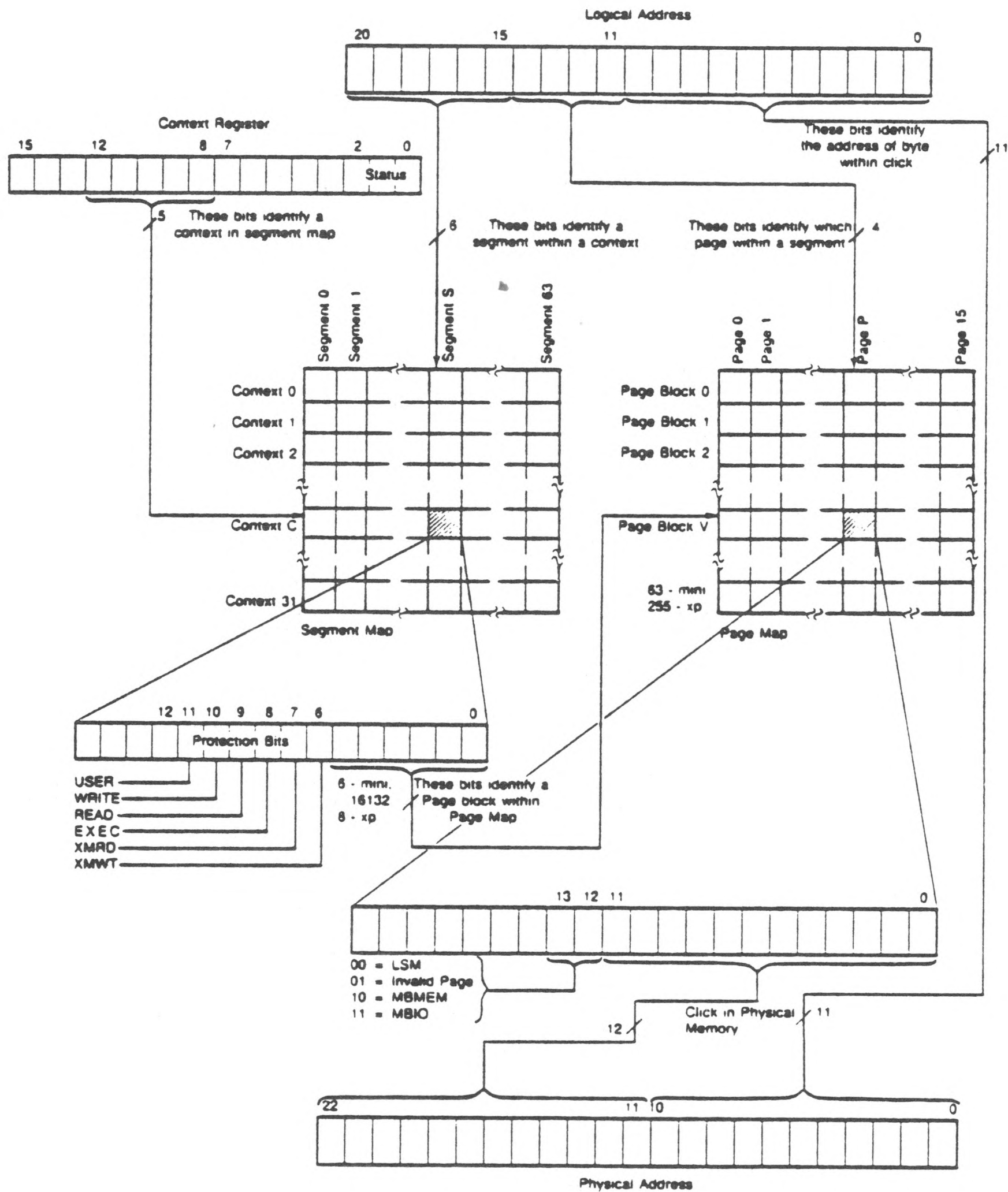


Figure 5. Memory Management Contexts

The page block number along with the middle significant bits in the logical address identify one cell in the page map. Each cell in the page map contains the actual address of a *click* of memory. In other words, each cell contains the most significant address bits of a physical memory location. The *click* address along with the least significant bits of the logical address make up the physical address.

Up to this point the MMU has not been an issue because all of kernel space is mapped identically into all contexts. For the 5000/20 and 5000/40, kernel space consists of logical addresses 0x00000 - 0x007FFF and 0x108000 - 0x1FFFFFF; segments 0, 33 - 63 in all contexts have the same values. For the 5000/50, kernel space consists of logical addresses 0xE00000 - 0xFFFFF. Refer to figure 6.

When the device being controlled transfers information into kernel space--for example, into the kernel buffers-- it doesn't matter how the non-kernel segments in context 31 are filled since they are never used. If, however, a user buffer is to be accessed in context 31, the buffer must be mapped into the user space of context 31 before DMA begins.

The virtual disk driver doesn't have a problem since it does not actually use DMA and therefore doesn't use context 31. A device driver, however, can temporarily execute code in context 31. This permits the simulation of DMA and demonstrates memory mapping concepts by extending the virtual disk driver example.

Mapping User Space Into Context 31

In order to map user space into context 31:

- Segments need to be reserved for use in context 31 for mapping of user space.

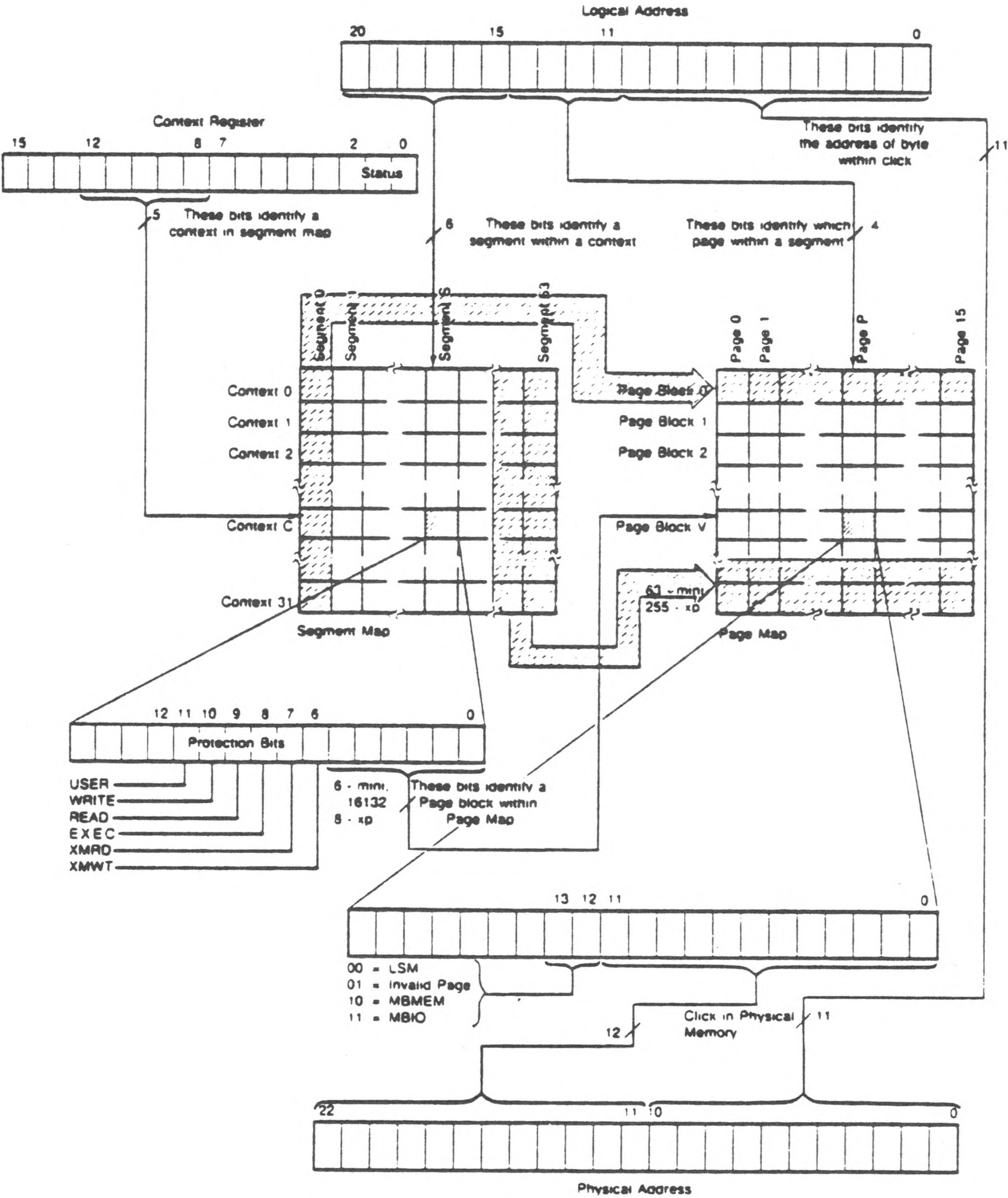


Figure 6. Memory Management Mapping

- A page block needs to be allocated for use by the driver.
- Each time physical I/O takes place, the correct physical addresses of user space must be initialized to map user space. Requests may need to be broken into smaller chunks if they are larger than the map space.

Segment Allocation

Segment allocation is part of the kernel configuration process. The bit called "required DMA resources" in the *master* file entry for the driver must be set. The *config* variable *dmanpb* is the number of page blocks and segments needed for each driver that requires DMA resources. The value of *dmanpb* is normally two. The external variable *<hp>n timer* has the number of DMA segments assigned to the driver. The external variable *<hp>l timer* is the logical address that identifies the segment reserved for the driver.

Reserving Page Blocks

Once, at system initialization, the page blocks needed by the driver are reserved. *Dmamalloc(k)* is used to reserve the page blocks.

NOTE: The number of bytes needed to map, not the number of page blocks needed, is the parameter for this routine.

Mapping User Space

Mapping user space into context 31 updates the MMU so that physical addresses corresponding to a user buffer can be accessed using logical addresses in context 31.

The kernel function call *dmapit(k)* brings together the following parameters to initialize the MMU.

- Logical address space (segments) allocated during configuration (*<up>lseg*).
- Pages allocated by *dmamalloc*
- Physical addresses of user space to be mapped.

Accessing logical addresses, starting at address *<hp>lseg* in context 31, addresses the same *clicks* in physical memory of the user buffer. The process of breaking up a large user buffer into smaller buffers and mapping those buffers interactively into context 31 is not a trivial process.

Virtual Disk Driver Init Definition

The initialization routine was added to reserve the page blocks needed. There are 32k bytes per page block. The value of *vdnpb* was two; a total of 64k bytes were requested.

Init Listing

```
/*-----  
* vdinit - initialize routine. Allocate page blocks from the page map  
*-----  
*/  
vdinit()  
{  
    vdpages = dmamalloc(VDMAPBYTES);  
    printf("vdpages: %x, vdnpb: %x, vdlseg: %x0, vdpages, vdnpb, vdlseg);  
}
```

Strategy Definition

A new strategy routine specifically for physical I/O has been added. This routine has all of the intelligence needed to map the I/O request into context 31 and to start the DMA transfer.

The user buffer is broken up into smaller pieces. DMA is then performed on each piece. Figure 7 shows the user buffer and the context 31 DMA map buffer starting at the address *vdldseg* and continuing for 64k bytes. The user buffer must be broken into pieces with these characteristics:

- An even number of physical disk blocks must be contained in the buffer.
- The maximum amount of space should be mapped.
- The DMA buffer size of 64k cannot be exceeded.

In the example, the number of mappable blocks in the user buffer can be calculated by:

$$(64k - clkoffset)/512$$

The number of bytes between the user buffer and the next lower *click* boundary is the value of *clkoffset*. In other words, the maximum number of bytes that could be mapped is 64k (128 blocks) if the user buffer happened to begin on a *click* boundary. If the user buffer doesn't begin on a *click* boundary, the number of bytes specified by *clkoffset* and the number of bytes in the partial block at the end of the map buffer are wasted (Figure 8).

The following C code initializes variables used to define the buffer. *Usrbuf* is the user buffer to transfer, *blkno* is the block number on disk, and *bp->b_resid* is the number of bytes left to transfer.

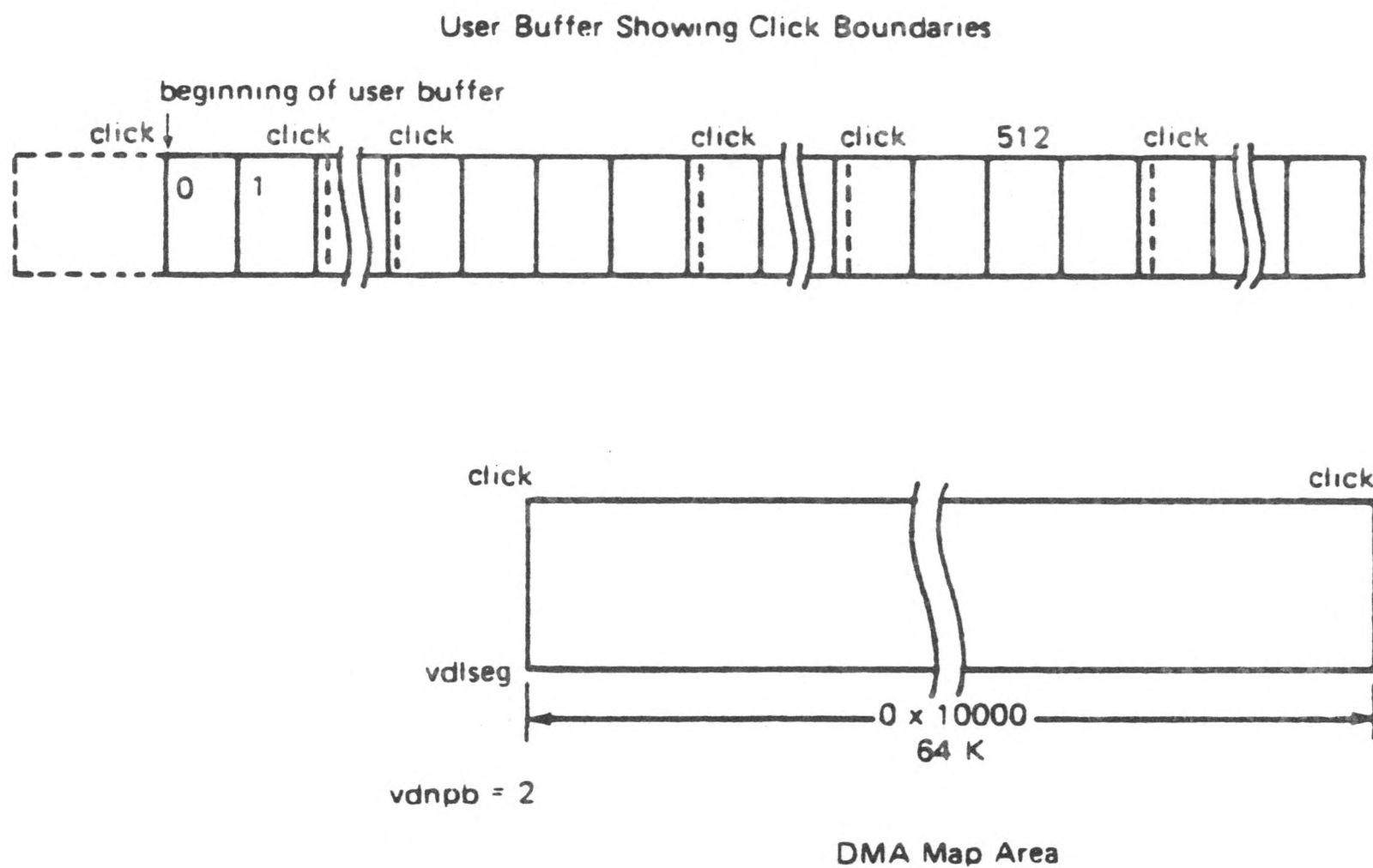


Figure 7. DMA Buffer/Map

```
usrbuf = bp->b_un.b_addr;    /* user buffer */
blkno = bp->b_blkno;         /* starting block number on disk */
bp->b_resid = bp->b_bcount;   /* number of bytes to transfer total */
```

The following code is the C source that performs the mapping calculations.

```
clkoffset = (int)usrbuf & CLKLSB; /* offset into first click */
/* byte count */
bcount = min((VDMAPBYTES - clkoffset), bp->b_resid) & VDPBLKMSB;
```

VDMAPBYTES is 64K. The DMA map area can be mapped using these variables:

```
dmapit(vdlseg, vdpages, usrbuf, bcount, bp->b_proc->p_spt);
```

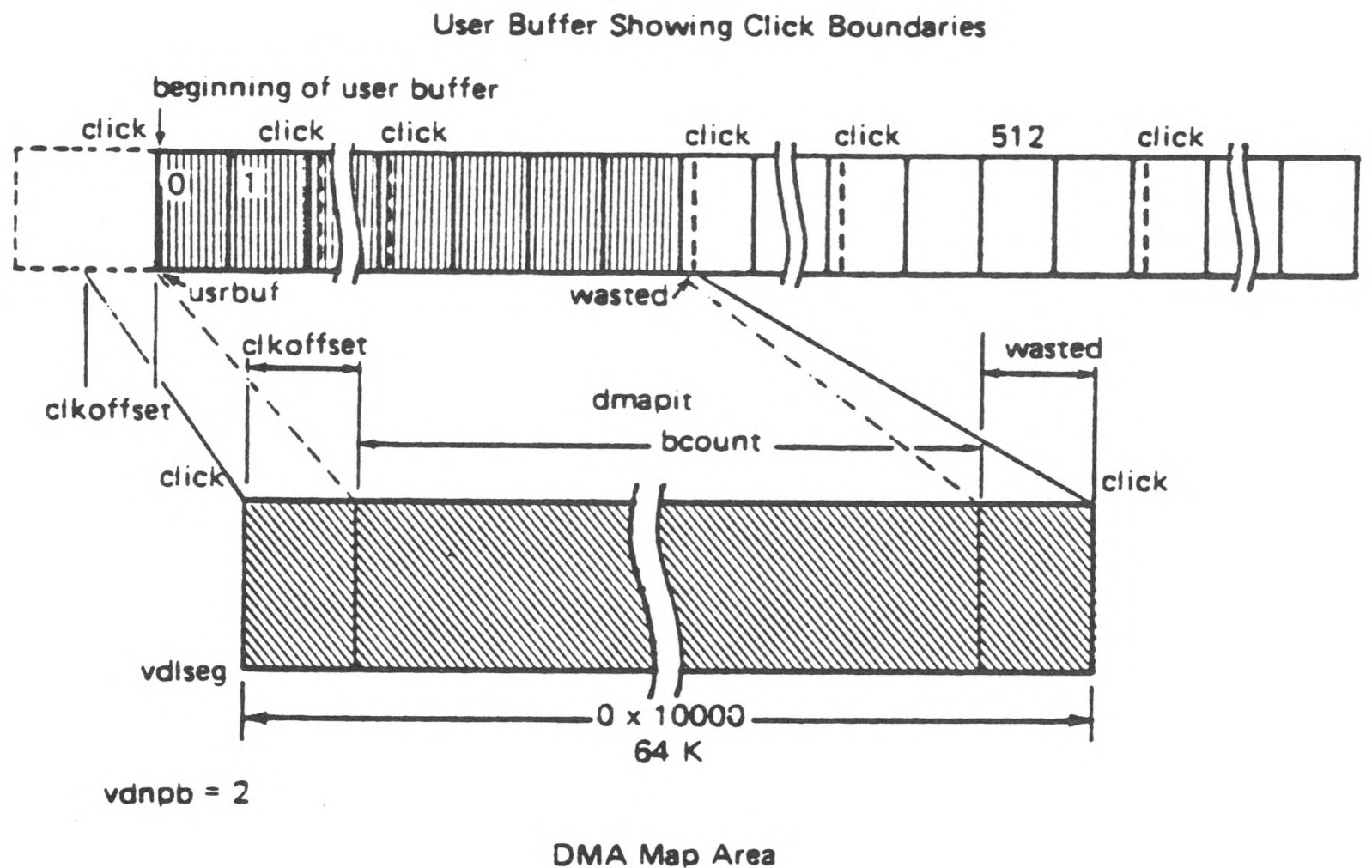



Figure 8. First Map

The function call *dmapit* performs these functions:

- The segment map in context 31 is initialized to the page blocks indicated by *vdpages*.
- The page map is filled with the *click* addresses identified by *usrbuf*, *bcount* and *bp->b_proc->p_spt*.

After this call has completed, the data in the first piece of the user buffer can also be accessed in context 31 by using addresses starting at *vdlseg*. The buffers and counters are all incremented and the process is repeated.

Figures 9 and 10 show the second and last maps.

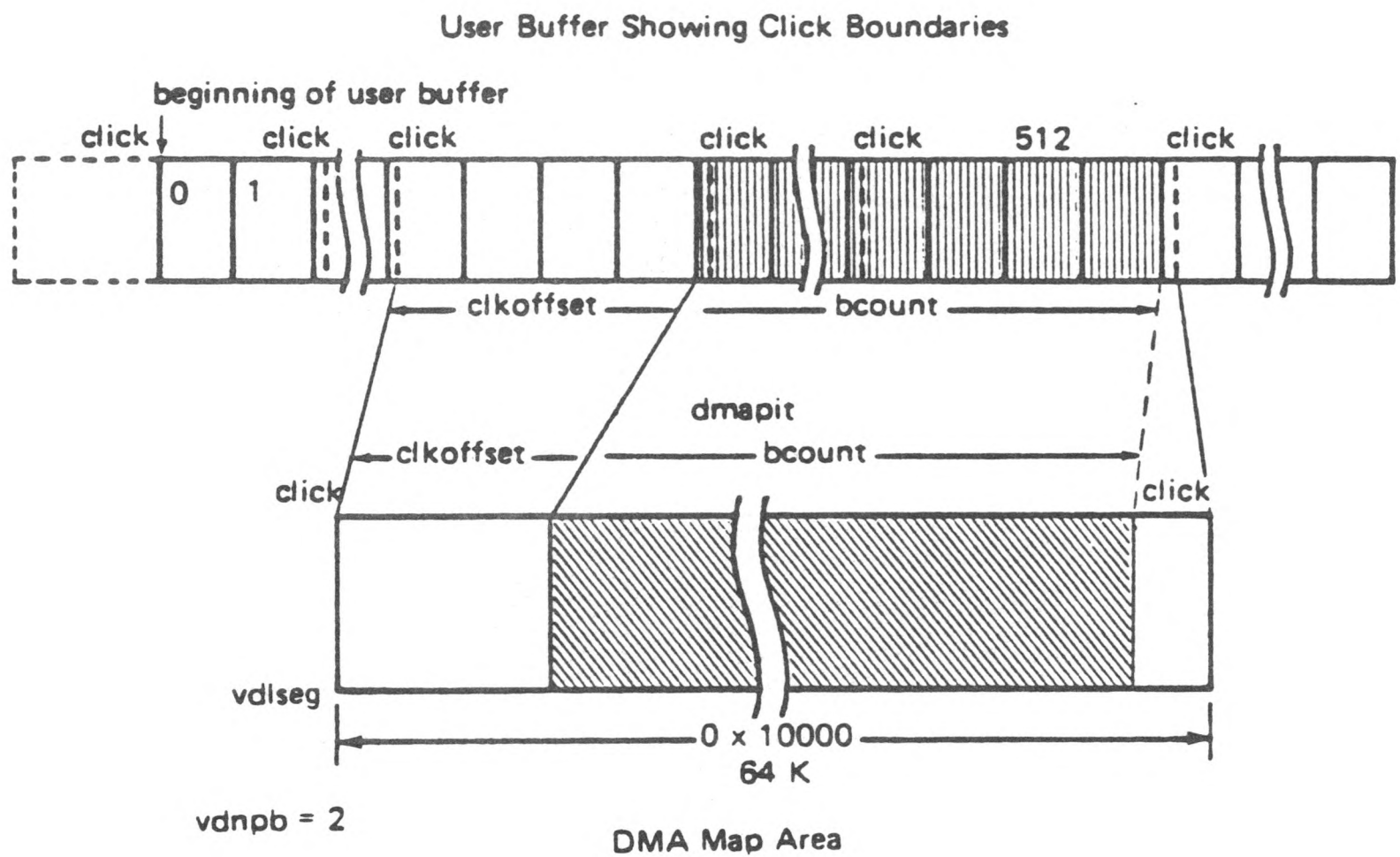


Figure 9. Second Map

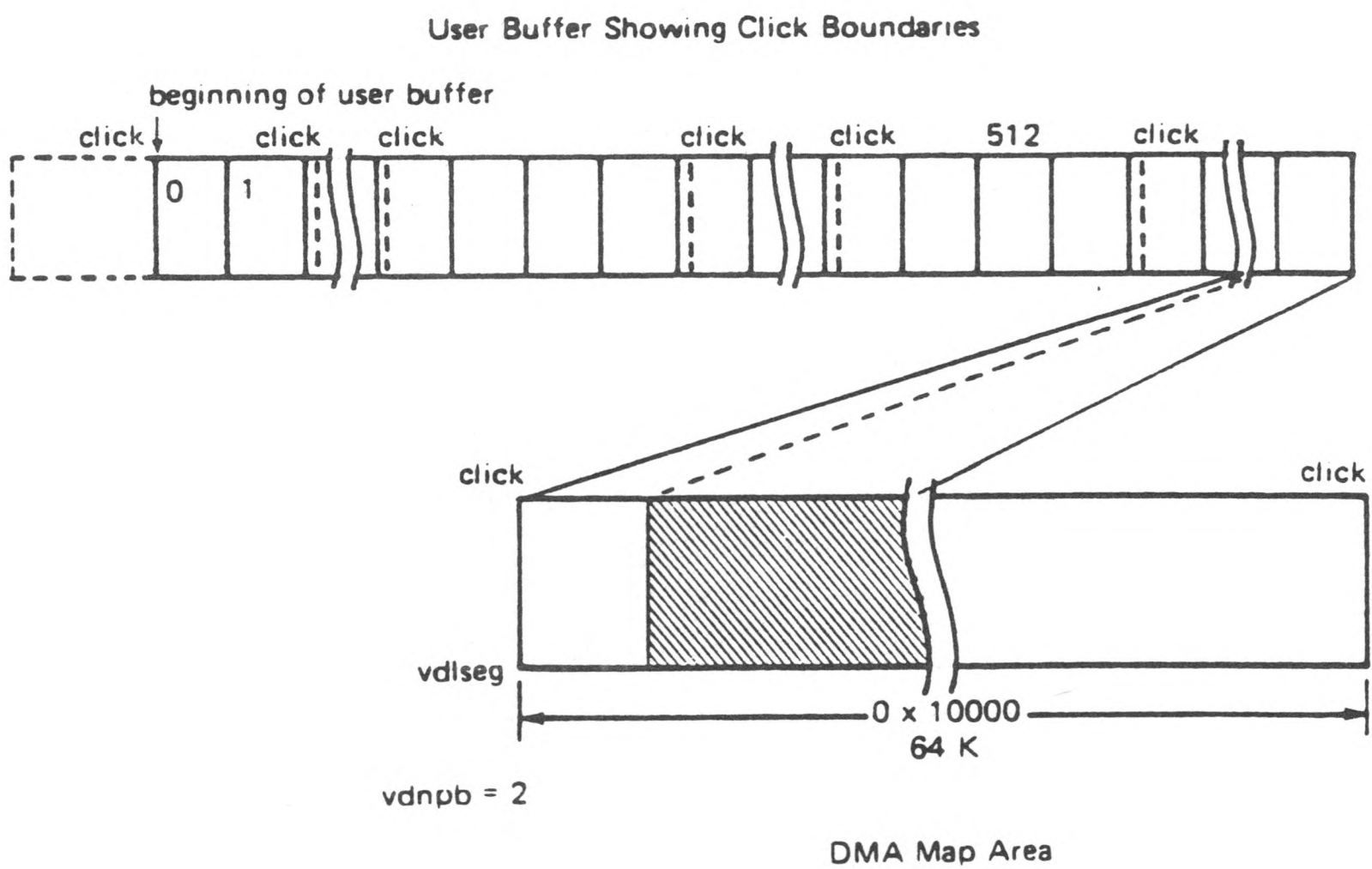


Figure 10. Last Map

PHYSICAL DEVICE IMPLEMENTATION

The following sections provide the code for the physical device implementation.

Read

```

/*-----
 * mmu constants
 *-----
 */
#define CLKLSB 0x7ff          /* click least significant bits */
#define CLKMSB 0xfffff800    /* block most significant bits */

/*-----
 * external references
 *-----
 */
extern int vd_cnt;          /* number of sub devices */
extern short vdnbp;         /* number of 32k dma segments reserved */
extern char * vdlseg;       /* address of first dma segment in context 31 */

#define VDMAPBYTES (vdnbp * 0x8000) /* number of bytes in the dma map area */

int vdpages;                /* page block pointer filled in vdinit */

/*-----
 * vdpstrat - physical io strategy using dma techniques
 *-----
 */
vdpstrat(bp)
struct buf * bp;            /* buffer pointer describing physical io */
{
    int clkoffset;          /* bytes at begining of first click that are skipped */
    int blkno;              /* current block number */
    int bcount;             /* number of bytes to transfer */
    char * mapbuf;          /* buffer mapped in context 31 */
    char * usrbuf;          /* buffer in user space */

    printf("vdpstrat: ");
    if (vdverify(bp))
    {
        usrbuf = bp->b_un.b_addr; /* user buffer */
        blkno = bp->b_blkno;       /* starting block number on disk */
        bp->b_resid = bp->b_bcount; /* number of bytes to transfer total */
        while (bp->b_resid >= VDPBLK && blkno < VDPBLKS)

```

Chapter 10

```
{
    clkoffset = (int)usrbuf & CLKLSB; /* offset into first click */
    /* byte count */
    bcount = min((VDMAPBYTES - clkoffset), bp->b_resid) & VDPBLKMSB;
    dmapit(vdlseg, vdpages, usrbuf, bcount, bp->b_proc->p_spt);
    mapbuf = (char *)((int) vdlseg | clkoffset); /* offset into dma map */
    startdma(mapbuf, blkno, bcount, bp->b_flags);
    blkno += bcount >> VDPBLKSHIFT; /* bcount div 512 */
    bp->b_resid -= bcount;
    usrbuf += bcount;
}
}
iodone(bp);
}

/*-----
 * startdma - do the dma transfer in context 31
 *-----
 */
vdcallrequest (bp_p)
struct buf ** bp_p;
{
    print2("vdcallrequest: *bp_p->b_un.b_addr %x0, (*bp_p)->b_un.b_addr);
    vdrequest(*bp_p);
}

startdma(mapbuf, blkno, bcount, flags)
char * mapbuf;
int blkno;
int bcount;
int flags;
{
    struct buf tempbuf;

    print5("startdma: mapbuf: %x, blkno: %x, bcount: %x, flags: %x0, mapbuf, blkno, bcount, flags);

    tempbuf.b_un.b_addr = mapbuf;
    tempbuf.b_flags = flags;
    tempbuf.b_blkno = blkno;
    tempbuf.b_bcount = bcount;
    cntxt31(vdcallrequest, &tempbuf);
}

struct buf vd_buf;

/*-----
 * vread - character device interface read routine
 *-----
 */
vread(minor_d)
```



```

dev_t minor_d;
{
physio(vdpstrat, &vd_buf, minor_d, B_READ);
}

/*-----
 * vdwrite - character device interface write routine
 *-----
 */
vdwrite(minor_d)
dev_t minor_d;
{
physio(vdpstrat, &vd_buf, minor_d, B_WRITE);
}

```

Complete Block and Physical Device Driver

```

/*-----
 * vd.c -- virtual disk driver
 *-----
 */

/*-----
 * include files
 *-----
 */
#include <sys/sysmacros.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/signal.h>
#include <sys/errno.h>
#include <sys/dir.h>
#include <sys/file.h>
#include <sys/user.h>
#include <sys/buf.h>
#include <sys/iobuf.h>
#include <sys/b0.h>
#include <sys/proc.h>

/*-----
 * global definitions
 *-----
 */
#define VDPBLK      0x200      /* physical block size in bytes */
#define VDPBLKMSB  0xfffffe00 /* physical block most sig bit mask */
#define VDPBLKLSB  0x1ff      /* physical block least sig bit mask */
#define VDPBLKSHIFT 0x09      /* power of 2 equal to VDPBLK */

```

```

#define VDPBLKS    0x100    /* number of physical blocks on disk */

struct vdblock
{
    char    data[VDPBLK];
};
struct vdblock vddisk[VDPBLKS];

struct buf vd_buf;          /* internal buffer */

/*-----
 * debugging routines
 *-----
 */
#define DEBUG
#ifdef DEBUG
    int vd_debug = 1;
#   define print1(a) vd_printf(a)
#   define print2(a,b) vd_printf(a,b)
#   define print3(a,b,c) vd_printf(a,b,c)
#   define print4(a,b,c,d) vd_printf(a,b,c,d)
#   define print5(a,b,c,d,e) vd_printf(a,b,c,d,e)
    vd_printf(a,b,c,d,e)
    {
        if (vd_debug) printf(a,b,c,d,e);
    }
#else
#   define print1(a)
#   define print2(a,b)
#   define print3(a,b,c)
#   define print4(a,b,c,d)
#   define print5(a,b,c,d,e)
#endif

/*-----
 * external references
 *-----
 */
extern int vd_cnt;          /* number of minor devices */

#define TRUE 1
#define FALSE 0

/*-----
 * vdstrategy - strategy routine
 *-----
 */
vdstrategy(bp)
struct buf *bp;
{

```

```

int physblk;

if (vdverify(bp))
    vdrequest(bp);
iodone(bp);
}

/*-----
 * vdverify - verify buffer pointer, return true to continue with the
 * request or false to stop processing
 *-----
 */
vdverify (bp)
struct buf * bp;
{
    if (bp->b_blkno == VDPBLKS) /* end of file, not really an error */
    {
        bp->b_resid = bp->b_bcount; /* number of bytes not yet transfered */
        return FALSE;
    }
    else if (bp->b_blkno > VDPBLKS) /* error: block number is to hi */
    {
        printf("error block number to high0);
        bp->b_resid = bp->b_bcount; /* number of bytes not yet transfered */
        bp->b_flags |= B_ERROR;
        return FALSE;
    }
    else
        return TRUE;
}

/*-----
 * vdrequest - fullfill a request
 *-----
 */
vdrequest(bp)
struct buf * bp;
{
    if (bp->b_flags & B_PHYS) /* physio call */
    {
        printf("physio %x0, bp->b_blkno);
        if (bp->b_flags & B_READ) /* read or write operation */
            copyout (&vddisk[bp->b_blkno], bp->b_un.b_addr, bp->b_bcount);
        else
            copyin (bp->b_un.b_addr, &vddisk[bp->b_blkno], bp->b_bcount);
    }
    else /* normal buffered io call */
    {
        printf("buffered %x0, bp->b_blkno);
        if (bp->b_flags & B_READ) /* read or write operation */

```

```

        blt (bp->b_un.b_addr, &vddisk[bp->b_blkno], bp->b_bcount);
    else
        blt (&vddisk[bp->b_blkno], bp->b_un.b_addr, bp->b_bcount);
    }

}

/*-----
 * mmu constants
 *-----
 */
#define CLKLSB 0x7ff          /* click least significant bits */
#define CLKMSB 0xfffff800    /* click most significant bits */

/*-----
 * external references
 *-----
 */
extern int vd_cnt;          /* number of sub devices */
extern short vdnpb;        /* number of 32k dma segments reserved */
extern char * vdlseg;      /* address of first dma segment in context 31 */

#define VDMAPBYTES (vdnpb * 0x8000) /* number of bytes in the dma map area */

int vdpages;              /* page block pointer filled in vdinit */

/*-----
 * vdpstrat - physical io strategy using dma techniques
 *-----
 */
vdpstrat(bp)
struct buf * bp;          /* buffer pointer describing physical io */
{
    int clkoffset;        /* bytes at beginning of first click that are skipped */
    int blkno;            /* current block number */
    int bcount;           /* number of bytes to transfer */
    char * mapbuf;        /* buffer mapped in context 31 */
    char * usrbuf;        /* buffer in user space */

    printf("vdpstrat: ");
    if (vdverify(bp))
    {
        usrbuf = bp->b_un.b_addr; /* user buffer */
        blkno = bp->b_blkno;      /* starting block number on disk */
        bp->b_resid = bp->b_bcount; /* number of bytes to transfer total */
        while (bp->b_resid >= VDPBLK && blkno < VDPBLKS)
        {
            clkoffset = (int)usrbuf & CLKLSB; /* offset into first click */
            /* byte count */
            bcount = min((VDMAPBYTES - clkoffset), bp->b_resid) & VDPBLKMSB;
            dmapit(vdlseg, vdpages, usrbuf, bcount, bp->b_proc->p_spt);
        }
    }
}

```



```

        mapbuf = (char *)((int) vdlseg + clkoffset); /* offset into dma map */
        startdma(mapbuf, blkno, bcount, bp->b_flags);
        blkno += bcount >> VDPBLKSHIFT; /* bcount div 512 */
        bp->b_resid -= bcount;
        usrbuf += bcount;
    }
}
iodone(bp);
}

/*-----
 * startdma - do the dma transfer in context 31
 *-----
 */
vdcallrequest (bp_p)
struct buf ** bp_p;
{
    print2("vdcallrequest: *bp_p->b_un.b_addr %x0, (*bp_p)->b_un.b_addr);
    vdrequest(*bp_p);
}

startdma(mapbuf, blkno, bcount, flags)
char * mapbuf;
int blkno;
int bcount;
int flags;
{
    struct buf tempbuf;

    print5("startdma: mapbuf: %x, blkno: %x, bcount: %x, flags: %x0, mapbuf, blkno, bcount, flags);

    tempbuf.b_un.b_addr = mapbuf;
    tempbuf.b_flags = flags;
    tempbuf.b_blkno = blkno;
    tempbuf.b_bcount = bcount;
    cntxt31(vdcallrequest, &tempbuf);
}

struct buf vd_buf;

/*-----
 * vread - character device interface read routine
 *-----
 */
vread(minor_d)
dev_t minor_d;
{
    physio(vdpstrat, &vd_buf, minor_d, B_READ);
}

```

Chapter 10

```
/*-----
 * vdwrite - character device interface write routine
 *-----
 */
vdwrite(minor_d)
dev_t minor_d;
{
    physio(vdpstrat, &vd_buf, minor_d, B_WRITE);
}

/*-----
 * vdinit - initialize routine. Allocate page blocks from the page map
 *-----
 */
vdinit()
{
    vdpages = dmamalloc(VDMAPBYTES);
    printf("vdpages: %x, vdnpb: %x, vdlseg: %x0, vdpages, vdnpb, vdlseg);
}

/*-----
 * vdclose - nothing
 *-----
 */
vdclose()
{
    print1("vdclose: not implemented0);
}

/*-----
 * vdprint - needed to get rid of undefined references
 *-----
 */
vdprint()
{
    print1("vdprint: not implemented0);
}

/*-----
 * vdioc1 - not implemented
 *-----
 */
vdioc1()
{
    print1("vdioc1: not implemented0);
}
```


Chapter 11.

Miscellaneous Procedures

POWER FAIL RECOVERY

When the power supply hardware detects a power failure, a power down interrupt is issued to inform the kernel. After the power down interrupt is issued, the kernel has a short amount of time to save the system registers and the contents of the time of day chip.

The return of external power initiates this sequence of events:

- The kernel restores the memory management unit.
- The kernel forces a level 1 interrupt to be pending.
- The kernel disables the *critical out* routines.
- The kernel returns from the down interrupt. This return to the point where the power failure occurred permits any critical code segments, which were interrupted by power down, to complete.
- The level 1 interrupt occurs and returns control to the kernel.
- The kernel enables the *critical out* routines.
- The kernel calls the driver power clear routines.
- After the last power clear routine has returned, the kernel returns from the level 1 interrupt and execution resumes.

Returning to the point of the down interrupt is done to permit any driver or kernel critical code segment to complete before the drivers are called to perform their

power fail recovery tasks. For example, consider the case where power down occurs in a disk driver's queuing routine. If this code is not permitted to run to completion, the state of the driver's queues is not known. If the driver's power clear routine manipulated these queues, they would become even more confused.

Critical segments must be protected by priority level of 1 or higher.

The *critical out* routines (*coutb*, *coutw*, *coutl*) are used by drivers to access on-board registers. Calls to the routines have the form

```
coutx (register_addr, data);
```

Where x is

b - If data is a byte.

w - If data is a word (16 bits).

l - If data is a long word (32 bits).

Before the return to the point of down, with the level 1 interrupt pending, the kernel disables the *critical out* routines. This prevents access to an uninitialized controller during this phase.

The power clear routines are driver and hardware specific. Power clear routines do whatever is necessary to restore the external devices to the state just before the power failure. This requires that the driver save information about the device state in a form that can be used by the power clear routine. For example, bus vectored devices provide an interrupt vector byte to the host as part of the interrupt cycle. The host uses this vector byte to calculate the interrupt vector assigned to the device. For maximum flexibility, most bus vectored devices allow the driver to pass the vector byte to the controller at run time. This is normally done only once, at system initialization, but it must also be done after a power failure.

BUS ERRORS AND USER MEMORY SPACE

Kernel modules, including drivers, should never access application data directly. Application programs often pass an incorrect address to the kernel in a system call. This usually happens as the result of some bug in the application program. Remember, a development system that tends to crash when applications are being debugged is not going to be popular. The basic rule that must guide any system module development is that users should not be able to do anything that can crash the system.

The kernel provides several routines that drivers can use to read or write application data. These routines protect the kernel against bus errors that, if not guarded against, would halt the system. Bus errors that occur when an application is running cause the application to stop, and a bus error when kernel code is executing causes a panic.

In all the examples given in this section, source and destination are pointers.

The general routine used to handle bus errors in all the routines, except *blt*, is *setjmp*. The kernel *setjmp* is a different routine than the library routine in section 3 of the programmer reference, but they are used in much the same way. The idea is to call *setjmp*, passing it a buffer where it can save the current state of the machine, and when it returns, make the access that is expected to cause an error. If it does, the kernel restores the state of the machine from the buffer that was passed to *setjmp* and execution resumes as a second return from *setjmp*. The two possible returns from *setjmp* are distinguished by the value returned by *setjmp*. If the return value is 0, then it is the first return. If it's not 0, then an error has occurred, and this is the second return.

Any code that uses *setjmp* must include *setjmp.h* and have a definition of *nofault*. In the example below, notice that the code saves the address of the old buffer and restores it. Failure to save and restore the old buffer address is likely to bring about a system crash that can be difficult to debug.

```
#include <sys/setjmp.h>
.
.
extern int *nofault;      /* Address of current jump buffer */
.
.
int iocheck (address)
    caddr_t address;
{
    jmp_buf jb;           /* jmb_buf: Jump buffer used to */
                          /* save the machine state. */
    int *saved_jb;        /* saved_jb: Used to save the */
                          /* address of the current buffer */
    int x;                /* x: Used to save the return */
                          /* value. */

    saved_jb = nofault;
    if (!setjmp (jb)) {
        /* Make jb the current jump buffer */
        nofault = jb;

        /* Access the suspected address */
    }
```



```

        x = *address;

        /* No error set x to good */
        x = 1;

    } else {
        /* Bus error happened */
        x = 0;
    }
    /* Restore the old jump buffer address */
    nofault = saved_jb;

    return(x);
}

```

The routine *fustr* is used to move null terminated strings between the kernel and application programs. It copies the source string to the destination string. The operation ends when the specified number of characters have been copied or when a null character is reached in the source string. If a null character is found, it is copied to the destination string. A 0 is returned if the copy completed successfully; a -1 is returned if a bus error happened while the copy was being done. As a note of caution, remember that this routine is designed to move ASCII strings; non-ASCII data is likely to have null characters that truncate the operation. For non-ASCII data, use *copyin* and *copyout*.

```
fustr (source, destination, number_of_characters);
```

For moving smaller quantities of data, there is another group of routines. The names all begin with either *fu* "Fetch User" or *su* "Set User" and end with the size of the data item they move. The sizes are byte and word -- a 4 byte quantity. The two routines *fuword* and *fubyte* return a data item read from the application's memory. All four routines return a -1 if an bus error is detected; *suword* and *subyte* return a 0 if no error is detected.


```
w = fuword (source);
c = fubyte (source);

suword (destination, source);
subyte (destination, source);
```

The function *iomove* is designed to be used in conjunction with the *read* and *write* system calls. It is controlled by a combination of its parameters and fields in the *u* structure. The three parameters are a kernel buffer, a number of bytes to be moved, and a flag that determines the direction of the transfer. This flag is either *B_READ* or *B_WRITE*. *B_READ* is taken to mean that *iomove* is being called from a *read* system call and that the transfer should be from the kernel to the application. The other flag, *B_WRITE*, means that the system call is a *write* and the move is from application memory to kernel memory.

```
/* Move data from the kernel to the user */
iomove (kernel_buffer_address, n_of_bytes, B_READ);

/* Move data from the user to the kernel */
iomove (kernel_buffer_address, n_of_bytes, B_WRITE);
```

The application's buffer address is in the field *u.u_base*. After *iomove* has finished, it increments *u.u_base* and *u.u_offset* by *n_of_bytes*. It also decrements *u.u_count* by *n_of_bytes*. There is one more thing -- the *u.u_segflg* should be 0. If it's a 1, the transfer is done without bus error protection. *Iomove* sets *u.u_error* to *EFAULT* if a bus error happens. Also, *iomove* uses *copyin* and *copyout* and they use *blt*.

Perhaps the two most flexible routines are *copyin* and *copyout*. *Copyin* copies into the kernel and *copyout* copies out to an application buffer. These routines return -1 if a bus error occurs and a 0 if successful.

```
copyin (source, destination, number_of_bytes);
copyout(source, destination, number_of_bytes);
```

Another useful routine is *blt*. This routine provides no protection against bus errors and so should not be used to copy between application memory and kernel memory. However, it is the fastest way to move blocks of data around within the kernel.

```
blt (destination, source, number_of_bytes);
```

ERROR LOGGING

The following sections describe the error logging structures and provide examples for error logging routines.

Communicating Errors To Application Processes

When a system call results in an error, the kernel or driver module that detects the error sets the *u_error* field in the *u* structure of the process that issued the system call. The valid settings of the *u_error* field are explained in the introduction to section 2 of the programmer reference. When the kernel returns from the system call with *u.u_error* set, it causes the system call to return a -1 and the *errno* variable is set equal to the *u_error* field. For example, the code fragments below describe an attempt by an application to open an invalid device.

```
/* Application Code */
.
.
#include <errno.h>
extern int errno;
.
.
if ((fd = open("/dev/x105", O_RDWR)) == -1) {
```

```

/* Open failed */
printf ("Unable to open /dev/x105. Errno = %d\n",
        errno);
/*
 * See perror in the programmer reference.
 */
perror("Unable to open /dev/x105.");
exit(1);
}
.
.
.

/* Kernel Code */
#include <sys/errno.h>
.
.
xlopen (dev)
    dev_t dev;
{
    /* X1 controller only supports subdevices 0, 1, 2, and 3 */
    if (minor(dev) > 3) {
        /* Invalid sub device number */
        /* ENXIO: Bad address or no device. */
        u.u_error = ENXIO;
        return;
    }
    .
    .
    .
}

```

Logging Hardware Events

Specific details of hardware errors are not reported to application processes. These details are stored in a log file (*/usr/adm/errfile*), and a system utility (*errpt(1)*) is provided for reading it. When a hardware error occurs, the driver creates an error record and places it in the in-memory error log. The in-memory error log is written out to the log file (*/usr/adm/errfile*).

Errpt reads the error records from */usr/adm/errfile*, interprets them, and prints them for a user to inspect. If the driver logs errors, add code to this utility to

print these error records.

Error Include Files

The error logging interface has three include files:

- */usr/include/sys/elog.h*
- */usr/include/sys/erec.h*
- */usr/include/sys/err.h*

The file *elog.h* contains structures that are used by block device drivers to collect statistics such as number of operations performed and number of unlogged errors. These statistics are logged whenever an error is logged. *Elog.h* also contains a set of definitions of major device numbers for logging purposes. These major numbers are different than the ones used in special device nodes. They are used as indexes and switch codes in *errpt*. Drivers that intend to log errors must add a major number to this list.

Erec.h contains definitions of standard error records, such as the error record that is logged when a power failure happens. There is a list in this file of error record types. These error record types are used in *errpt* to invoke the correct routine to print a record. If a driver intends to log a new error record type, then a new type code must be added to this list.

The last file is *err.h*, which contains structures that define the error log.

Character Device Error Logging

Character device error logging requires three steps.

1. Get space in the error log for the driver's error record.
2. Fill in the fields of the error record with the information that describes the problem.
3. Place the record in the log.

Drivers get space in the error log by using *geteslot*. This routine accepts the size of the record to be logged and returns a pointer to a buffer large enough to hold it. If no space is available in the error log, *geteslot* returns a null pointer.

```
struct errhdr *geteslot (size)
    int size;
{}
```

The driver fills the error record by redefining the pointer returned by *geteslot* as a pointer to a driver error record and then loading the fields according to the type of the error.

An error is logged by passing its address and type to *puterec*. *Puterec* places the time and the error type into an error header structure that precedes the user defined error record, and then logs the whole structure. The error type mentioned above is the type defined by the user and added to the list in *erec.h*.

The example below shows a driver logging a checksum error. Notice that the driver keeps a count of unlogged errors and logs that number when an error is successfully logged. The *makedev* macro builds device numbers from major and minor numbers. It is defined in *sysmacros.h*. Also, the major number XYZ0 is the one added to the list in *elog.h*.

It's good practice to place developer defined structures in include files. The driver and *errpt* can share the same include file.

```
/* Error Include File xyzerr.h */
/* Error codes */
#define CKSUMERR      (1)
.
.
struct xyzerr {
    dev_t edev;
    int ecode;
    int eulogd;
    .
    .
    .
};
```

Now the driver code:

```
/* Driver Source Code */
#include <sys/sysmacros.h>
#include <sys/eelog.h>
#include <sys/erec.h>
#include <sys/err.h>
#include <sys/xyzerr.h>
.
.
struct xyzerr *ep;
int unlogged = 0;
.
.
if ((ep = (struct xyzerr *)geteslot(sizeof(struct xyz))) != NULL) {
    ep->edev = makedev (XYZ0, logical_unit);
    ep->ecode = CKSUMERR;
    ep->eulogd = unlogged;
    .
    .
    puterec (ep, XYZtyp);
} else {
    ++unlogged;
}
```

Block Device Driver Error Logging

Error logging in block device drivers is more structured than error logging for character devices. Block device driver developers must add a major number to the list in *elog.h*, but they don't need to add a error record type to the list in *erec.h*. A standard error record type has been defined for block device drivers.

Information logged for block devices comes from the subdevice's queue header and from the buffer header for the I/O request that caused the error. Block device error logging routines assume that the first buffer header in the queue caused the error.

The first *open* for the subdevice must create the fields in the queue header. Remember, the queue header is defined in *iobuf.h*. The first field in the queue header is *b_dev*. The driver makes the value for the *b_dev* field by combining the error log major number, from the list in *elog.h*, with the subdevice number. The macro *makedev* can be used to do this.

The second field, *io_stp*, is a pointer to the subdevice's I/O statistics block. The keeping of I/O statistics by block devices is explained fully in the next section.

The last two fields are closely related. They define the address and size of a block of data that the driver developer wants placed in the log. The field *io_addr* contains the address of the block and *io_nreg* contains the number of 16-bit quantities that make up the block.

This mechanism was originally intended to log device registers. The present 5000 series disk drivers use it to log their I/O control blocks.

These I/O control blocks are data blocks in memory that describe the I/O operation to the controller.

The block device error logging interface contains two routines. *Fmtberr* creates an error record and prepares it for logging; *logberr* enters the record created by *fmtberr* into the log. Drivers also call *fmtberr* to log retries.

Fmtberr is called once when an error is detected and once for each retry. The parameters passed to *fmtberr* are the address of the queue header for the subdevice that experienced the error and the physical block number of the first block in the partition (see the section on partitions).

Logberr is called when the driver is ready to log the error. This happens when the driver's error recovery efforts end either successfully or otherwise. Two parameters are passed to *logberr*, the address of the queue header for the device that had the error and a flag that specifies the disposition of the error. If the flag is a 0, the error is logged as recovered; if the flag is a 1, the error is logged as unrecovered.

Block device drivers set a bit in the global variable *blkacty* whenever they are accessing the bus. This variable is logged by *fmtberr* to give information about simultaneous bus activity. Each driver is assigned the bit that corresponds to the driver's error log major number. Drivers should set this bit just before issuing a command to their controller and clear it when they receive an interrupt.


```
/* Set the bit */
blkacty |= (1 << XYZ0);
.
.
/* Clear the bit */
blkacty &= (1 << XYZ0);
```

Block Device I/O Statistics

Block device drivers collect statistics that are used to calculate estimates of error rates and I/O system performance. The data collected includes:

- Number of I/O operations done.
- Number of non-I/O operations done.
- Number of blocks transferred.
- Number of unlogged errors.
- Amount of time that the controller was busy.
- Amount of time between an operation being received by the driver and its completion.

The structure used to collect statistics is defined in *elog.h*; its name is *iotime*. Each subdevice's statistics are kept in its element of an array of *iotime* structures.

```
struct iotime xyzstat[N_of_SUBDEVs];
```

When a subdevice is opened for the first time, the field *io_stp* in its queue header must be made to point to the *iostat* field in the subdevice's *iotime* structure.

```
xyzutab[minor(dev)].io_stp = &xyzstat[minor(dev)].ios;
```

The *iostat* structure contains three fields. The first field, *io_ops*, is used to count the number of I/O operations the device has done. The second field, *io_misc*, is used to count non-I/O operations. The third field is not used by the driver directly. *Fmtberr*

uses *io_unlog* to count errors that it could not log because of a lack of error log space. The *ios* field is logged by *fmtberr*.

The remaining fields of the *iotime* structure are: *io_bcmt*, *io_act*, and *io_resp*. The *io_bcmt* field is used to total up the number of 512 byte blocks transferred to or from the subdevice. The kernel keeps a global variable named *lbolt*. *Lbolt* contains a count of the number of ticks of the system clock since the system was booted. Therefore, *lbolt* divided by *HZ* gives the number of seconds since the system was booted. The constant *HZ* is defined in *param.h*.

The *io_act* field is used keep a total of the time that the controller is busy. The driver does this by setting the field *io_start* in the subdevice's queue header equal to *lbolt* when a command is issued to the controller. When the command is completed, the driver subtracts the value of *io_start* from the current value of *lbolt* and adds the result to *io_act*. The last field is used to keep a total of the time between a I/O request being received by the driver and the request being completed. This is accomplished by saving *lbolt* in the *b_start* field of the buffer header when the request is received. When the request has been completed, just before the driver calls *iodone*, *b_start* is subtracted from *lbolt* and the result is added to *io_resp*.

```
/* In the strategy routine */
bp->b_start = lbolt;
.
.
.
/* In the routine that issues the command to the controller */
xyzutab[minor(bp->b_dev)].io_start = lbolt;
.
.
.
/* In the interrupt service routine */
sdn = minor(bp->b_dev); /* Subdevice number */
zyzstat[sdn].io_act += lbolt - xyzutab[sdn].io_start;
.
```

```
.  
.  
/* In the routine that releases the completed I/O request */  
xyzstat[minor(bp->b_dev)].io_resp += 1bolt - bp->b_start;  
iodone (bp);
```

Diagnostic System Call

The in-service diagnostic system call accepts two parameters: a null-terminated string containing the name of the device, and a pointer to a structure containing information that describes the diagnostic request. Each driver must define its own structure.

```
struct xlarg xl_arg;  
.  
.  
inserv ("h801", &xl_arg);  
.  
.
```

At the driver level, the interface consists of an array of pointers to device names and the inservice diagnostic routine. The name of the array of pointers is *nam*, preceded by the handler prefix specified in the *master* file (see Chapter 3). The last entry in the array is a null pointer. The example below is taken from the system's 5.25-inch disk driver. Notice that all possible devices and all possible variant spellings are represented in the array. The kernel uses this array to identify the driver that handles the device named in the inservice diagnostic system call.

```
char *wdnam[ ] = {  
    "h501",  
    "h502",  
    "h501",  
    "h502",  
    (char *)0};
```


The inservice diagnostic interface routine in the driver is named *xyzdiag* (xyz is handler prefix). It receives the two parameters that were passed to the inservice diagnostic system call. The device name string has already been copied by the kernel into the kernel environment, but the driver must copy in the diagnostic structure. Remember to use *copyin* and *copyout*.

```

    xldiag (devnme, diagstructure)
        char devname[];
        structure xlarg *diagstructure;
    {
    }

```

The main implementation problem is how to gain control of the hardware. Diagnostic requests are received asynchronously with regular I/O requests. The driver needs to have some mechanism for halting the processing of regular I/O operations, doing the diagnostic, and resuming regular I/O processing. In the system's disk drivers, this is accomplished by having the diagnostic interface routine queue a special buffer header and go to sleep to wait for the diagnostic to be completed. The special buffer header is defined within the driver and is not part of the kernel's buffer pool. When the start routine reaches that buffer header, it invokes a diagnostic operation routine to do the requested diagnostic operation. When the diagnostic operation is completed, the diagnostic operation routine wakes up the diagnostic interface routine, clears the driver busy flag, and calls the start routine to resume processing of regular I/O. If the diagnostic operation causes an interrupt, then the interrupt service routine must invoke the diagnostic operation routines.

Appendix A.

HPSIO Edited Functional Specification

CONTROLLER PORT DEFINITIONS

Port Assignments

Port Address*	Port Width	Definition
M + 0	4	Mailbox Register
M + 0	2	Mailbox Diagnostics Status
M + 4	2	Interrupt Vector Register
M + 6	2	Soft Arbiter/Int. Status Reg. **

*:
M = Board base address. This address is defined as the firmware default address combined with the specific board strapping.

***:
The Soft Arbiter/Interrupt Status Register can be accessed by a byte read at M + 7 or a word read at M + 6. (The upper byte of the word is valid.)

Mailbox Register - 32 bits

- Mail box is addressable using word or long word references. No byte operations.
- Writing to M + 2 interrupts HPSIO local processor.
- Reading "M + 2" clears interrupt from HPSIO local processor.
- Reading or writing "M" has no affect on interrupt.

- Access to mail box register must be coordinated with HPSIO local processor through use of soft arbiter.
- Intended to be used to pass addresses of IOPBs.
- First read after controller diagnostics shows diagnostics status.

Mailbox Register Diagnostics Status - 16 bits

Bits	Definition
0	Channel 0 Error Indicator
1	Channel 1 Error Indicator
2	Channel 2 Error Indicator
3	Channel 3 Error Indicator
4	Channel 4 Error Indicator
5	Channel 5 Error Indicator
6	Channel 6 Error Indicator
7	Channel 7 Error Indicator
8-11	Level-0 Test Running
12	Level-0 Completion Status
13-15	Critical Board Error Code

Channel x Error Indicators

- 0 = No error
- 1 = Channel degraded

Level-0 Test Running

If Bit 12 = 1, then these four bits show the HPSIO Level-0 test currently running.

Level-0 Completion Status

- 0 = Controller Level-0's not running
- 1 = Controller Level-0's in progress

Critical Board Error Code

- 0 = No error
- 1 = ROM checksum error
- 2 = RAM stuck bit/address decode error
- 3 = Mailbox register stuck bit error
- 4 = All channels degraded error

Interrupt Vector Register - 16 Bits

Bits	Definition
0-7	Vector to be supplied to PMC when HPSIO generates an interrupt to <i>Multibus</i> ; R/W from <i>Multibus</i> . HPSIO processor has no access to this.
8-10	Interrupt level assigned to HPSIO when it generates <i>Multibus</i> interrupts; should be selected as one of the "Bus Vectored" levels of the PMC and is read/write from <i>Multibus</i> . HPSIO processor has no access.
11	Defined to enable (=0) or disable (=1) interrupts from HPSIO processor to <i>Multibus</i> . R/W from <i>Multibus</i> . HPSIO processor has no access.
12-15	Reserved. Should be zero

Soft Arbiter/Interrupt Status Register - 8 Bits

Bits	Definition
0	= 1 if HPSIO processsor has pending interrupt from PMC write to "M + 2" mail box register. Cleared when HPSIO processor reads "M +2" mail box register.
1	= 1 if PMC has pending interrupt from HPSIO processor write to "M+2" mailbox register. Cleared when PMC reads "M+2" mailbox register.
2-5	Reserved. No meaning.
6	"ACK 1" function for the software arbiter; controls access to the mail box register. May be read by HPSIO Processor and from

Multibus but only written by HPSIO processor.

- 7 "REQ2" function for the software mail box arbiter. Readable by both local processor and from *Multibus*; written only from *Multibus*.

NOTE: D0 + D1 simultaneously = 1 is an error condition which signals a failure to follow soft arbiter protocol.

CIOPB STRUCTURE

Structure Definition

Mnemonic	Bytes	Description
FUNC	1	Function Code
STAT	1	Return Status
CHAN	1	Channel number
CBAUD	1	Trans. Rate; Signal Control
CMODE1	1	Parity Type; Char. Length
CMODE2	1	No. Stop Bits; Oper. Mode
CSTAT	1	Line Status
ICNTRL	1	Interrupt Control/SRQ
DSTAT	1	Character Status
DCHAR	1	Received Character
DSTAT	1	Character Status
BUFAD	4	Buffer Start Address
BUFLEN	2	Buffer Length
Hex Length	<u>10</u>	

CIOPB Field Definitions

FUNC:

Hex Value	Description
1	Channel Initialization
2	Character Acknowledgement
3	Configure Interrupt
4	Output
5	Reserved

STAT:

See LIOPB definition for STAT field

CHAN:

DUART channel number.

Valid numbers range from 0 to 7.

CBAUD:

Bits	Definition
0-3	BAUD Rate
4	Break Control
5	Enable Input Control
6	RTS Control
7	DTR Control

BAUD Rate

0 = 75bps

1 = 110

2 = 134

3 = 150

4 = 300

5 = 600

6 = 1200

7 = 2000

8 = 2400

9 = 4800

A(hex) = 1800

B(hex) = 9600

C(hex) = 19200

Break Control

0 = Normal

1 = Force Break

Enable Input Control

0 = Disable Input

1 = Enable Input

RTS Control

0 = Force RTS Output Low (Off)

1 = Force RTS Output High (On)

DTR Control

0 = Force DTR Output Low (Off)

1 = Force DTR Output High (On)

CMODE1:

Bits	Definition
0-1	Character Length
2	Parity Type
4	Parity Enable

Character Length

0 = 5 bit character

1 = 6 bit character

2 = 7 bit character

3 = 8 bit character

Parity Type

0 = Even parity

1 = Odd parity

Parity Enable

0 = Enables parity

1 = Disable parity

CMODE2:

Bits	Definition
0-3	Stop Bit Length
4-5	Reserved (should be zero)
6-7	Operational Mode

Stop Bit Length

7 = 1 Stop Bit

F(hex) = 2 Stop Bits

Operational Mode

0 = Non-echo

1 = Auto-echo

2 = Local Loopback

3 = External Loopback

CSTAT:

Bits	Definition
0	DSR State
1	DCD State
2	CTS State
3-7	Reserved (should be zero)

xxx State

0 = Signal xxx Low Status (Off)

1 = Signal xxx High Status (On)

ICNTRL:

Bits	Definition
0	Receiver Interrupt Enable
1	Input Line change Interrupt Enable
2	Receiver SRQ (Host read only)
3	Input Line Change SRQ (Host read only)

Receiver Interrupt Enable

- 0 = Disable "Character Available" Interrupt
- 1 = Enable "Character Available" interrupt

Input Line Change Interrupt

- 0 = Disable Delta Line Change Interrupt
- 1 = Enable Delta Line Change Interrupt

Receiver Service Request

- 0 = No service request
- 1 = Service request for "Data Character Available"

Input Line Change Service Request

- 0 = No service request
- 1 = Service request for Delta line change status

DSTAT:

Bits	Definition
0-3	Reserved (should be zero)
4	Overrun status
5	Parity status
6	Framing status
7	Break status

Overrun Status

- 0 = No overrun error
- 1 = Overrun error

Parity Status

- 0 = No parity error
- 1 = Parity error

Framing Status

- 0 = No framing error
- 1 = Framing error

Break Status

0 = No break condition

1 = Break condition (not necessarily an error)

DCHAR:

This field is used to store the data character input (through a DUART) to system memory. The transmission status of this character is in the DSTAT field.

BUFAD:

System memory logical address of the output buffer with an output function.

BUFLEN:

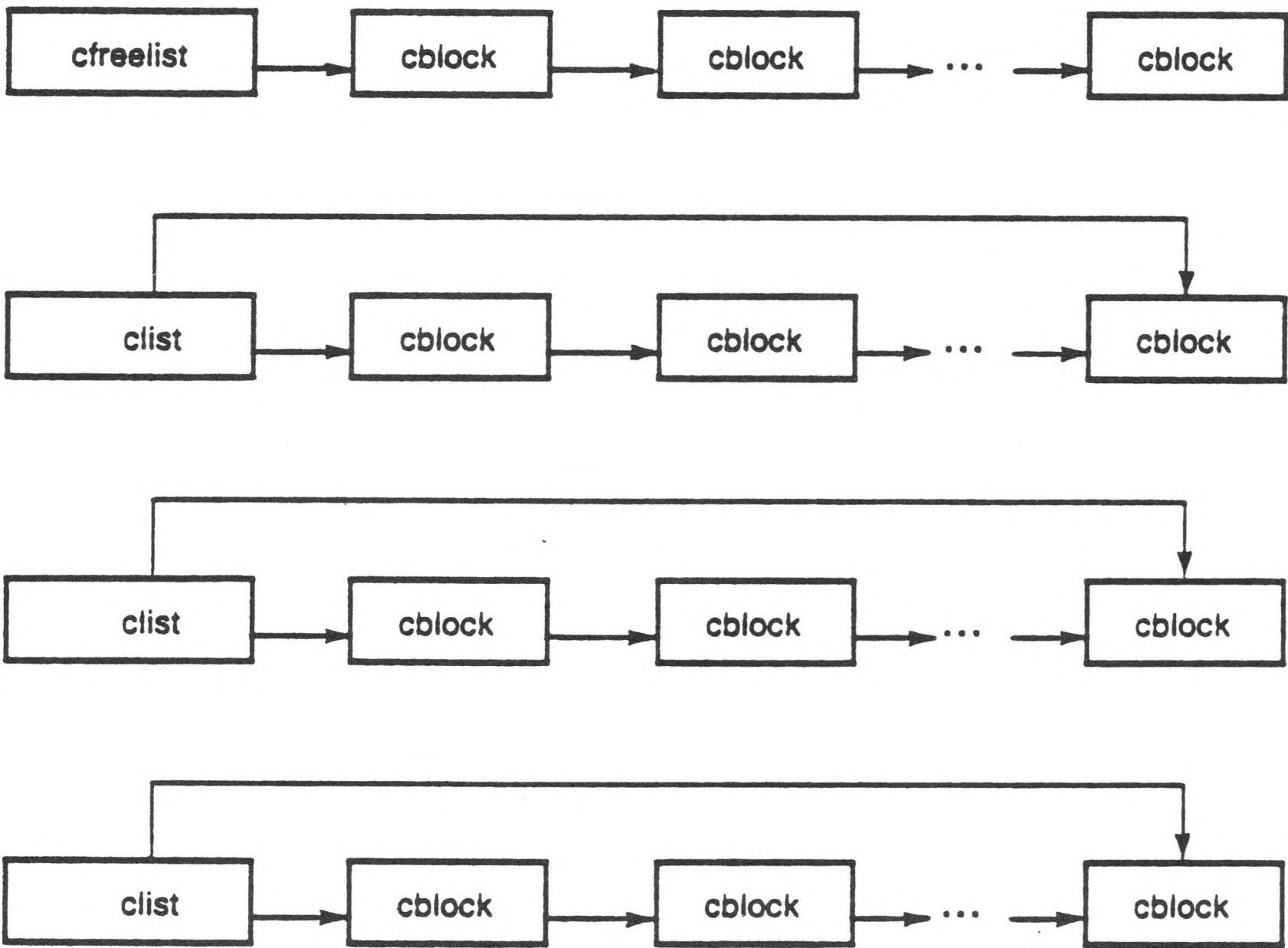
Length in bytes of the output buffer specified by the BUFAD field.

Appendix B.

Clists

DEVICE DRIVER CLISTS

Clists are queues of characters used by the kernel. *Clists* consist of one or more *cblocks*. Each *cblock* holds up to 64 characters. A free list, called *cfreelist*, is maintained to keep track of available *cblocks*.



CLIST DEFINITIONS

These definitions are found in the file */usr/include/sys/tty.h*.

```
/*-----
 * A clist structure is the head of a linked list queue of characters.
 * The routines getc* and putc* manipulate these structures.
 *-----
 */

struct clist {
    int c_cc;          /* character count */
    struct cblock *c_cf; /* pointer to first */
    struct cblock *c_cl; /* pointer to last */
};

/*-----
 * The structure of a clist block
 *-----
 */
#define CLSIZE 64
struct cblock {
    struct cblock *c_next;
    char    c_first;
    char    c_last;
    char    c_data[CLSIZE];
};

/*-----
 * designed just for cfreelist
 *-----
 */
struct chead {
    struct cblock *c_next;
    int c_size;
    int c_flag;
};
extern struct chead cfreelist;
```

CBLOCKS AND CLISTS

Each *cblock* contains:

c_next

Pointer to the next *cblock* in the list.

c_first

Index of next character in the *cblock* to be read.

c_last

Index of next character in the *cblock* to be written.

c_data

Index of an array of 64 characters.

Each *clist* consists of:

c_cc

The total number of (unread) characters in the *clist*.

c_cf

Pointer to the first *cblock* in this *clist*.

c_cl

Pointer to the last *cblock* in this *clist*.

CBLOCK FREE LIST

A free list is maintained to keep track of available *cblocks*, for fast allocation to a *clist*. The number of *cblocks* reserved is the *clists* configuration parameter. The *cfreelist* structure heads the free list, and contains:

c_next

Pointer to the first available *cblock*.

c_size

Number of characters per *cblock* (64 for this system).

c_flag

If set, processes are waiting for a *cblock* (wakeup will be issued when one is freed).

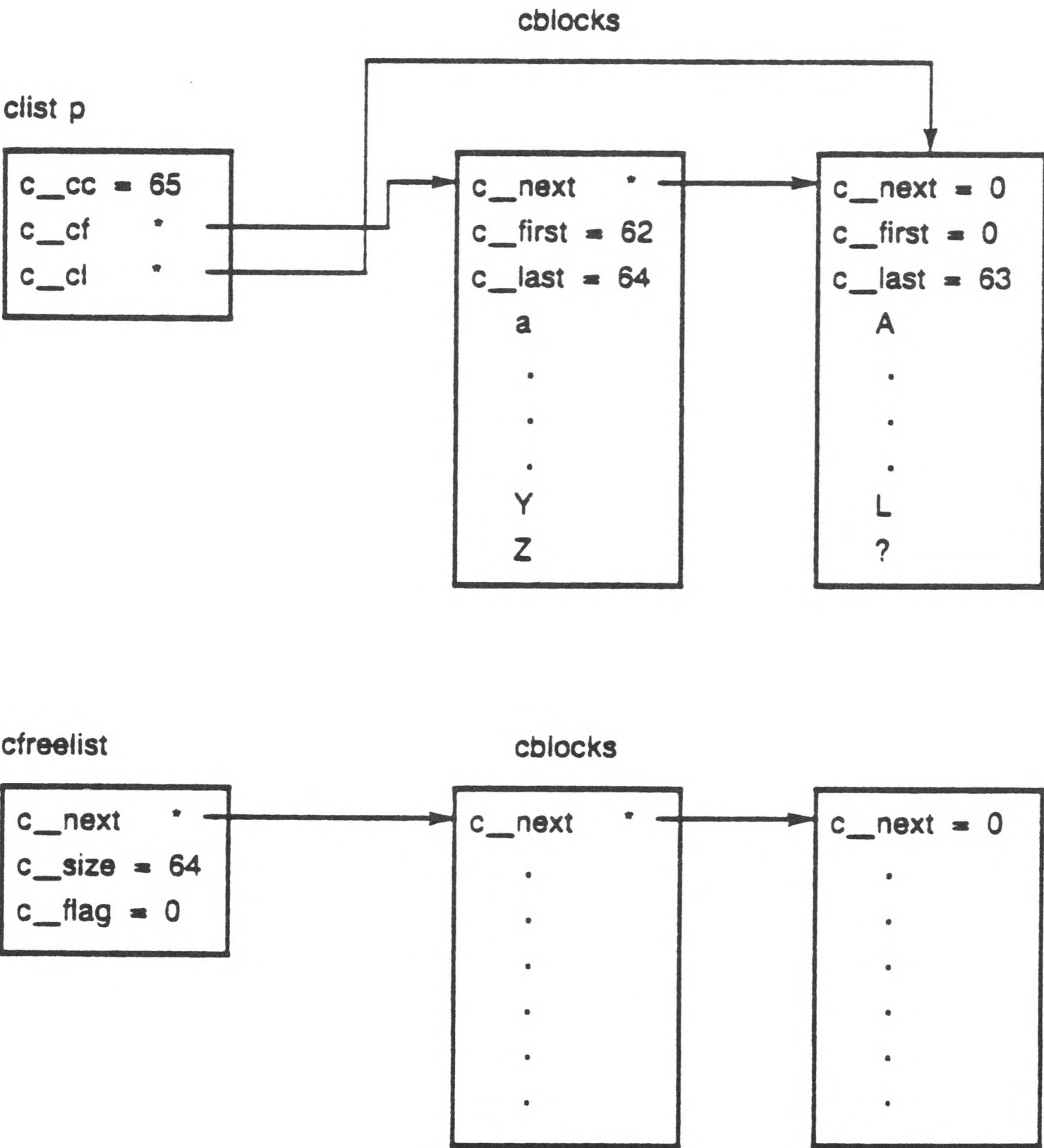
The *cblocks* on the free list are singly-linked using the *c_next* field.

CHARACTER BUFFERED I/O DIAGRAMS

The following diagrams show the general relationship between *cblocks*, a *clist*, and the free list for the kernel calls that influence these structures and buffers.

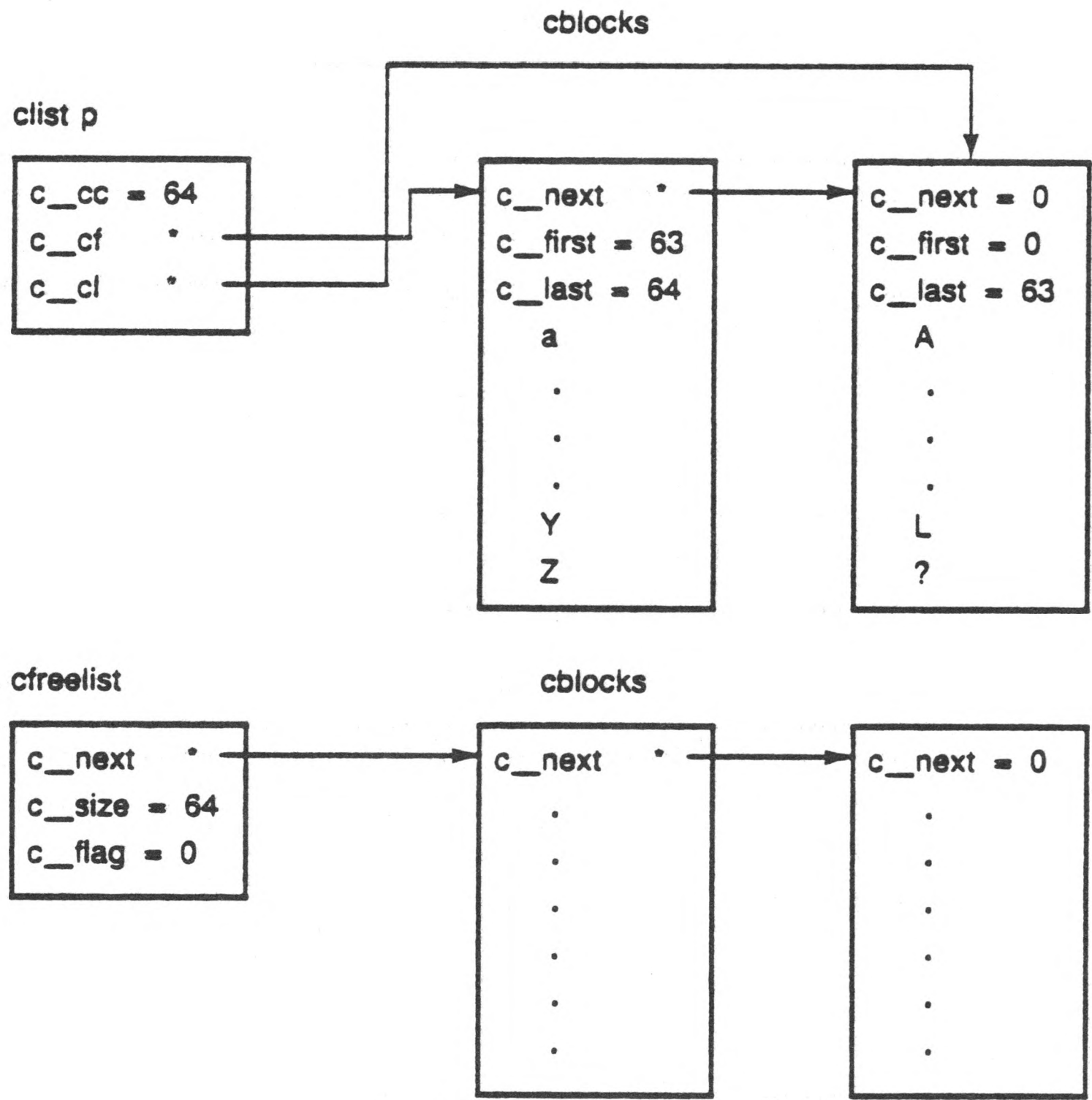
getc

Initial state of clist p and cfreelist



NOTE: The characters in a *cblock* are referenced by an index between 0 and 63.

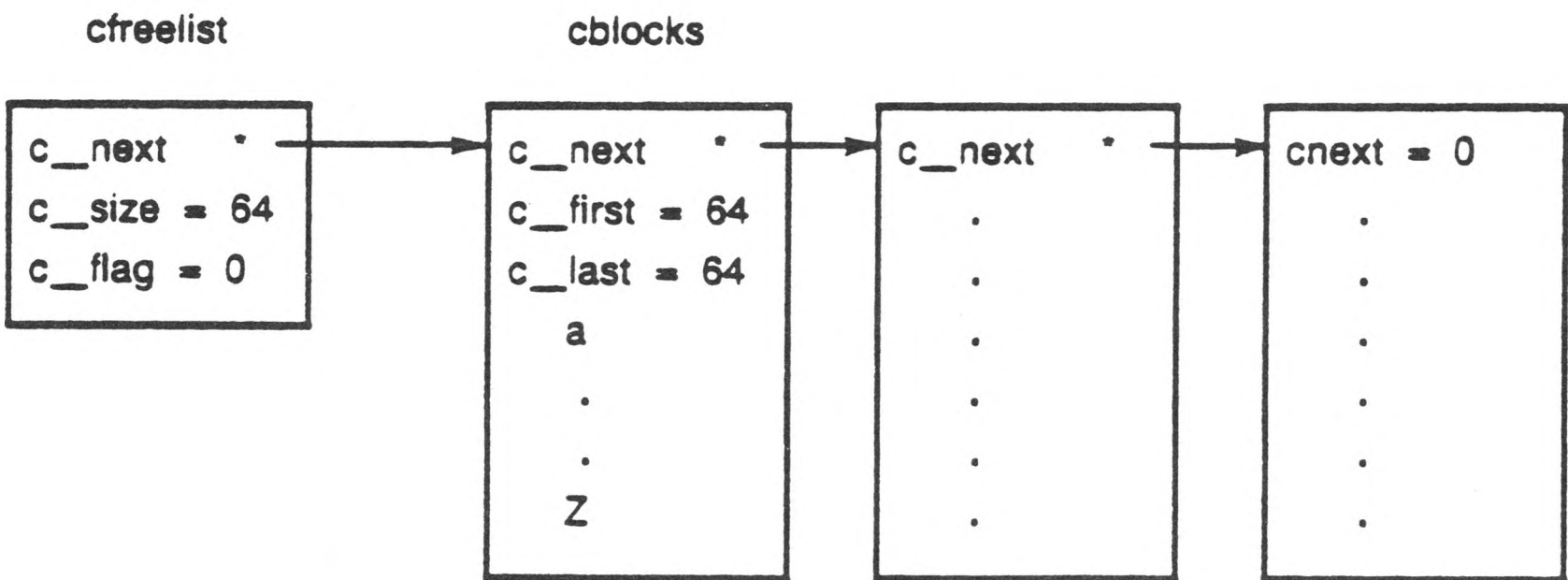
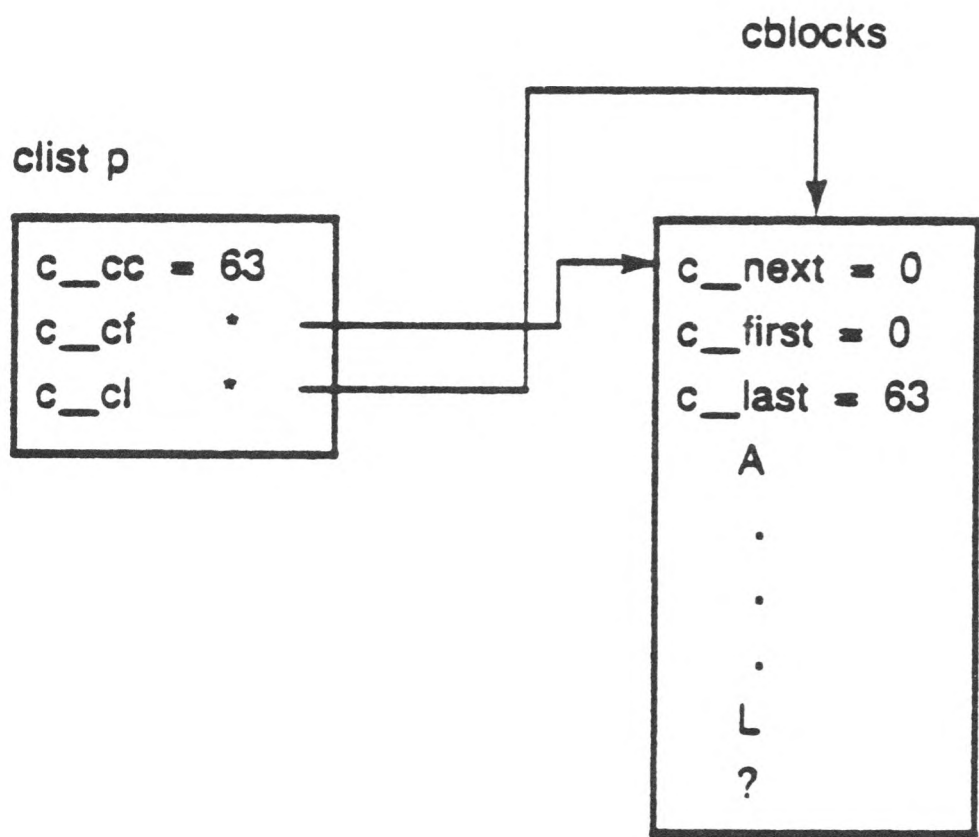
End of first getc operation on the clist p



The character 'Y' is returned.

NOTE: The `c_first` field of the `cblock` is incremented to 63 to indicate the next character to read, and the number of characters in the `clist` itself is decremented by 1.

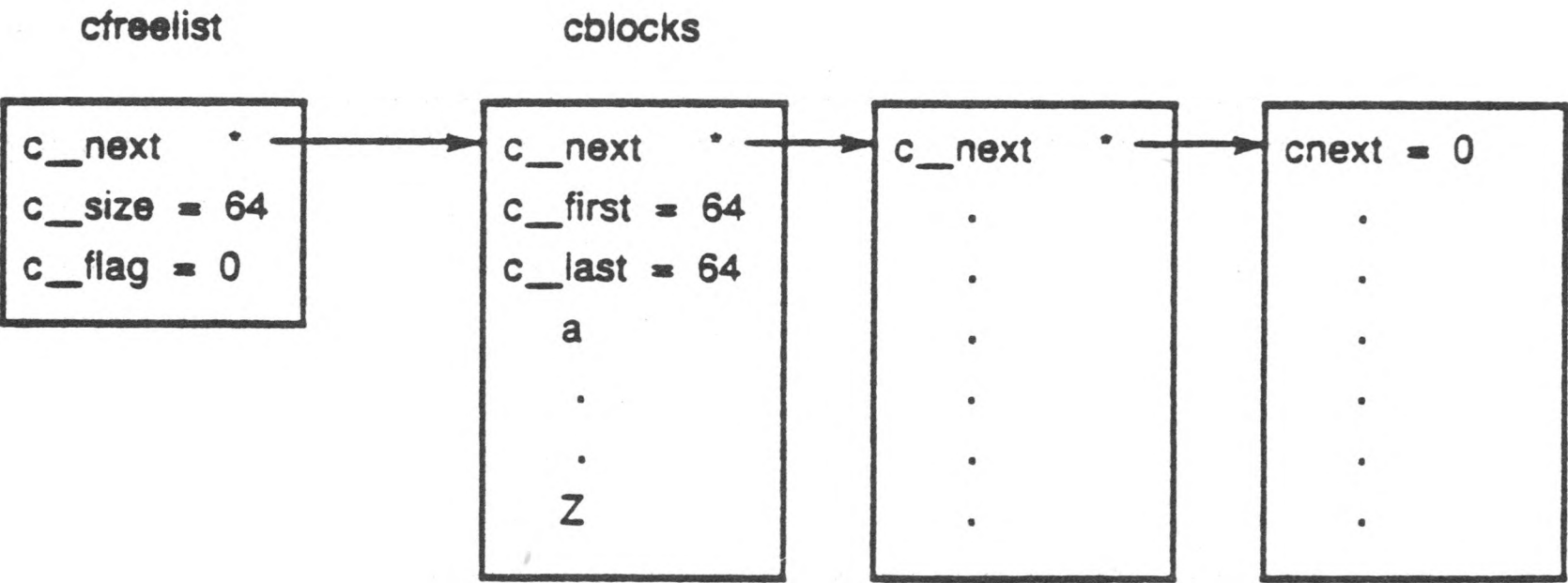
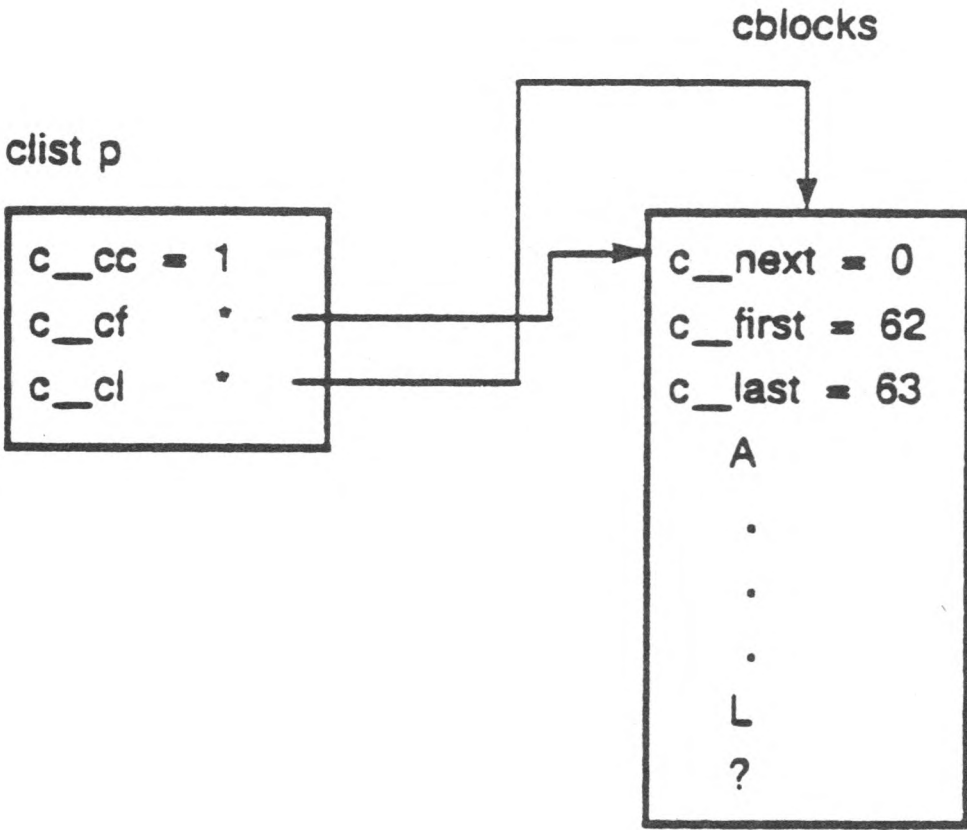
End of second getc operations on the clist p



The character 'Z' is returned.

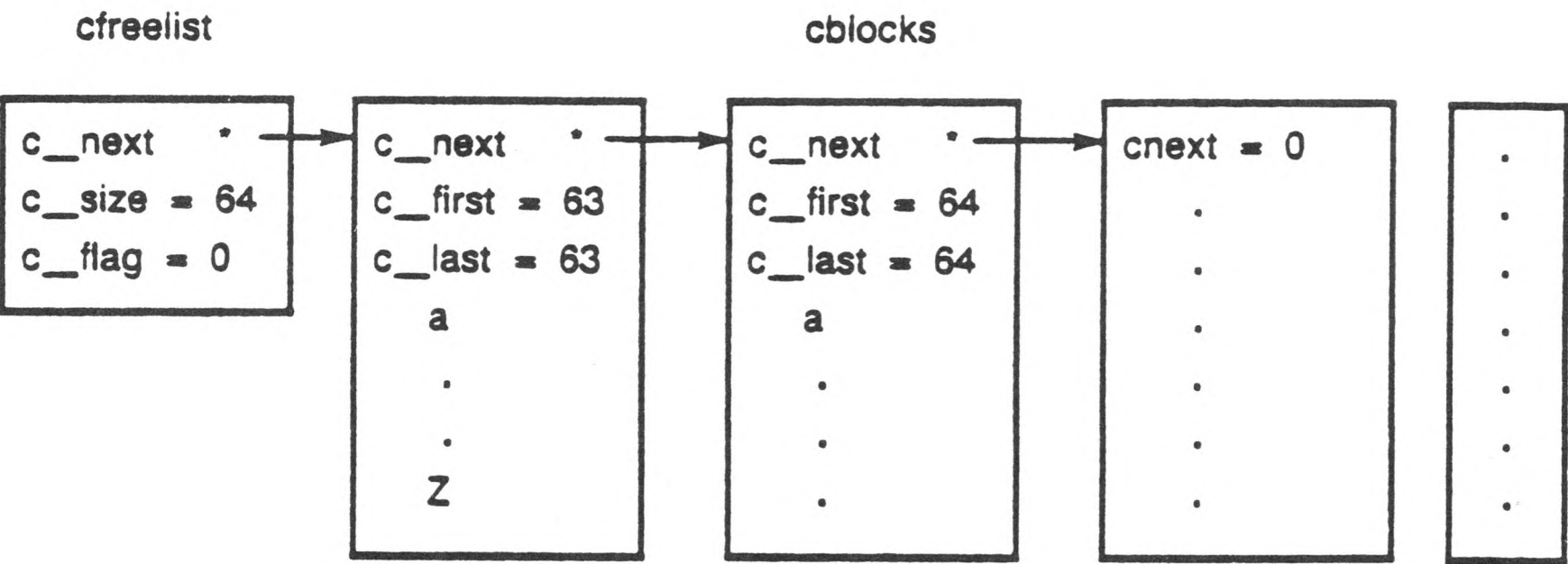
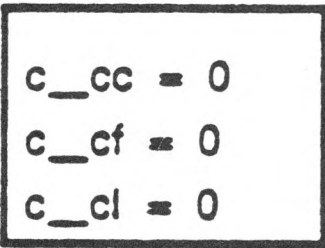
NOTE: Because the entire *cblock* has been read, it is returned to the free list. Both *c_cf* and *c_cl* point to the only *cblock* in the *clist*, and *c_cc* is decremented.

End of many getc operations on the clist p



End of next getc (clist is empty)

clist p



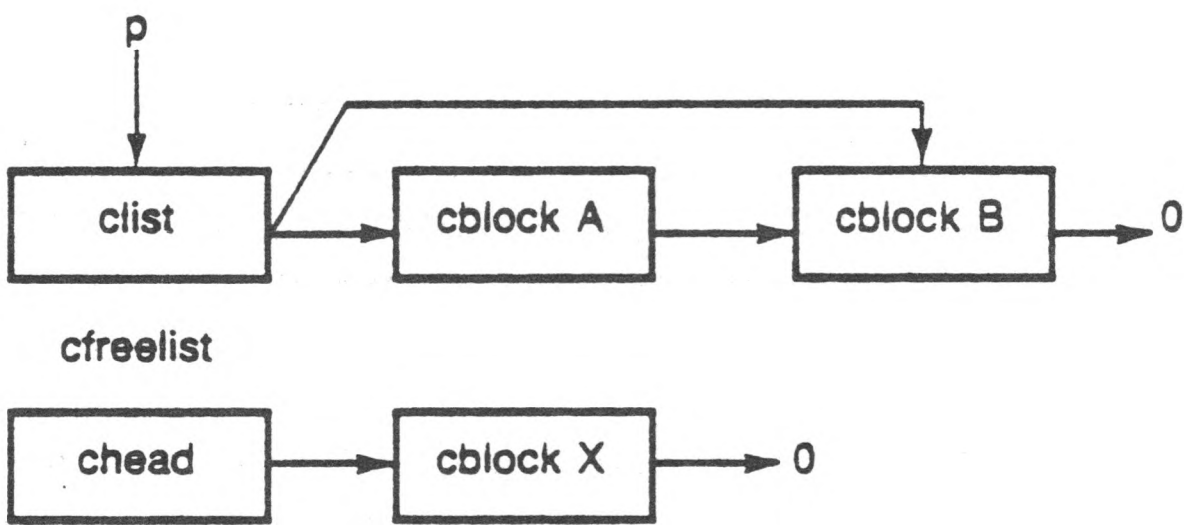
The character 'L' is returned.

NOTE: Because the entire *cblock* has been read, it is returned to the free list. Both *c_cf* and *c_cl* have the value 0 which indicates no *cblocks* on list.

The *getc* (p) operation returns a value of -1.

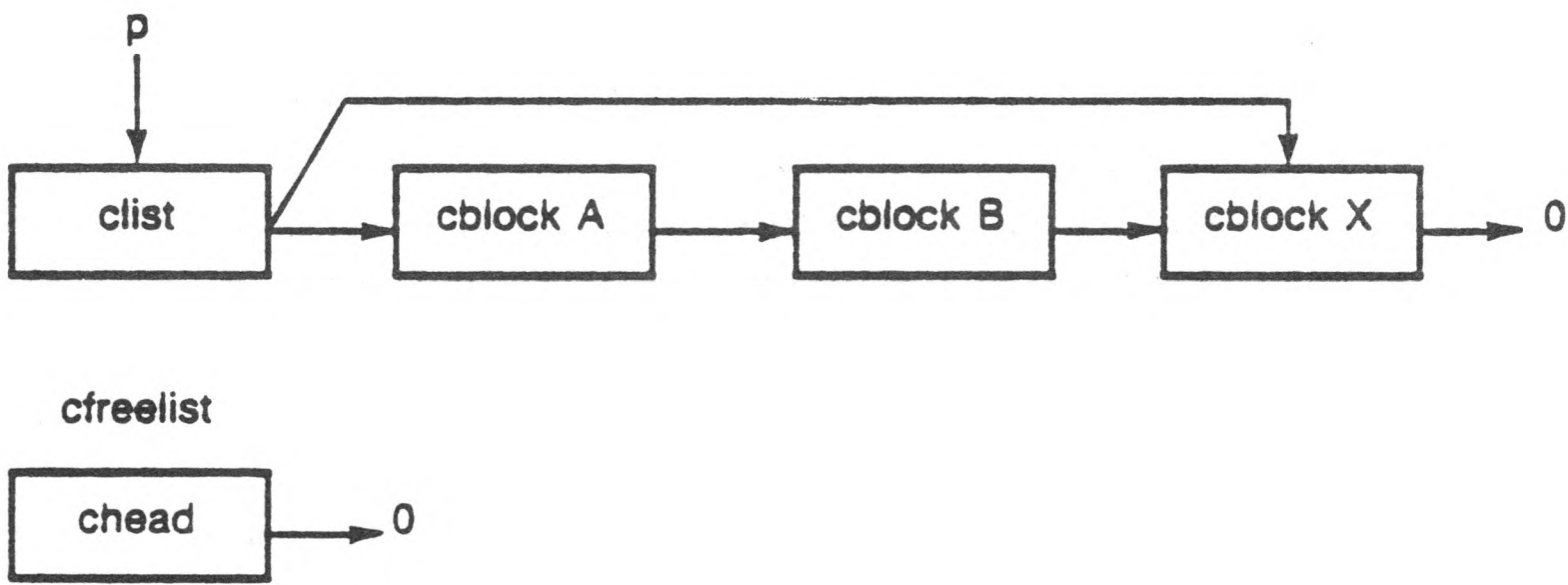
putc

Initial state of clist p and cfreelist



Each time `putc(c, p)` is called, the character is added to the last *cblock* in the *clist* p.

cblock added to clist

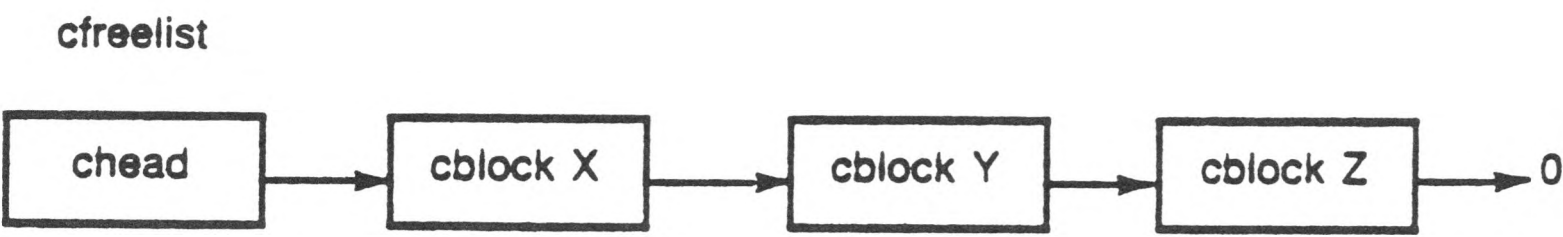


When the last *cblock* becomes full, `putc` adds another block to the *clist*. In the example *cblock* X is added.

When `putc` fails because no more blocks are available on *cfreelist*, the the value -1 is returned rather than zero.

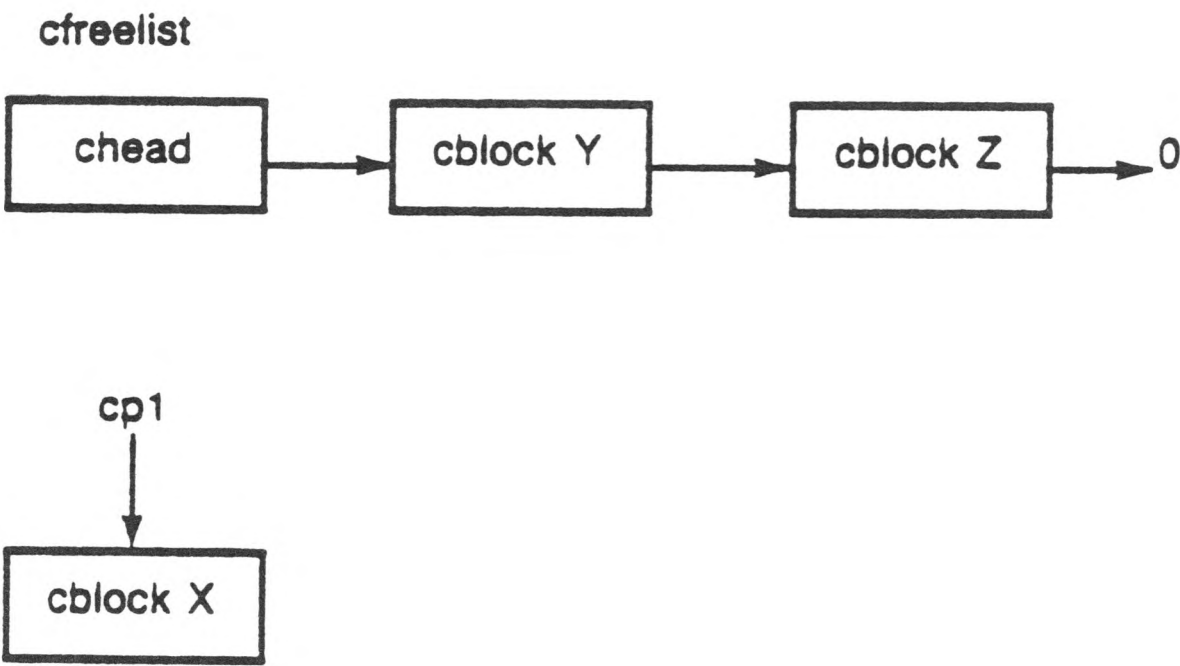
getc

Initial state of cfreelist

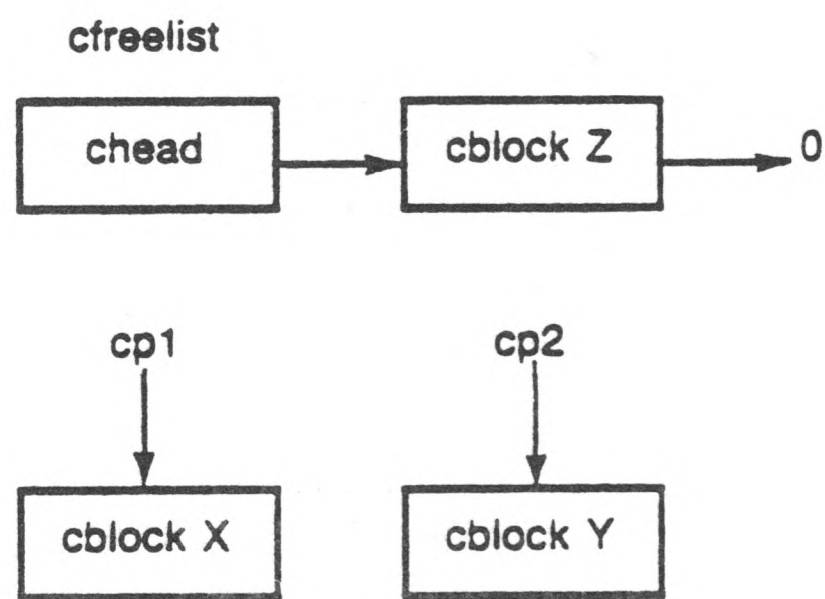


Each time *getc()* is called , an element on *cfreelist* is removed and its address is returned.

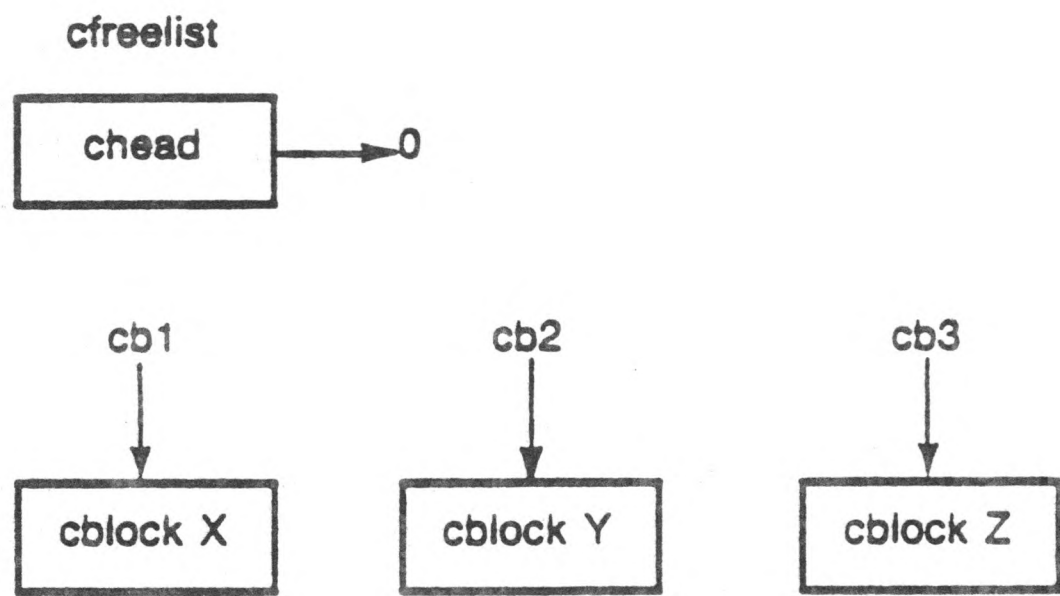
Result of cp1 = getc();



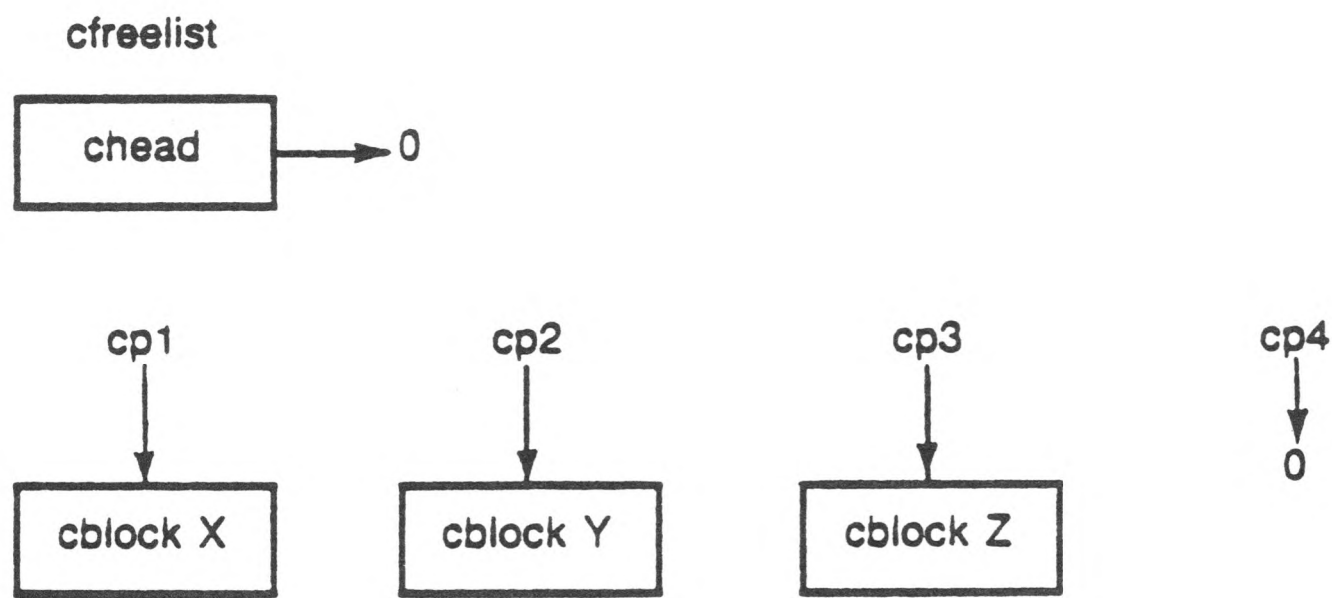
Result of cp2 = getcf() ;



Result of cp3 = getcf() ;



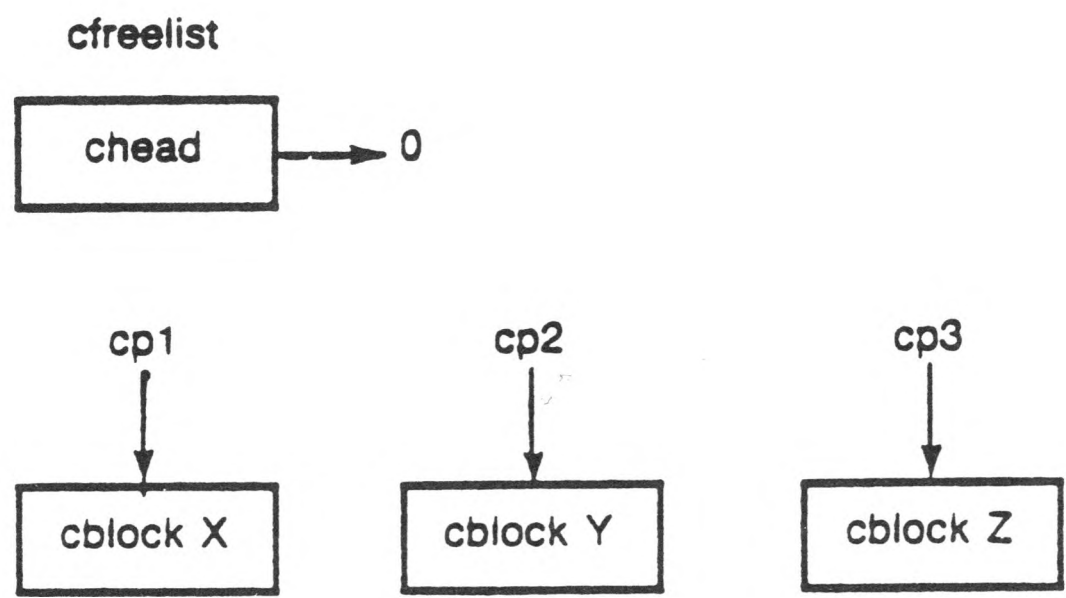
Result of `cp4 = getcf()` ;



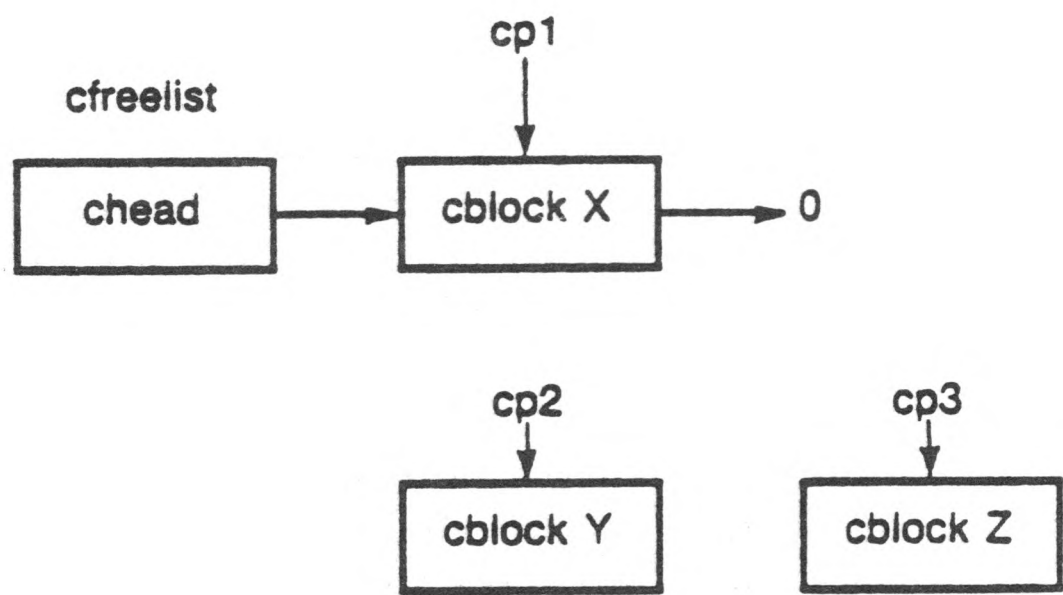
The value 0 is returned if the *cfreelist* is empty.

`putcf`

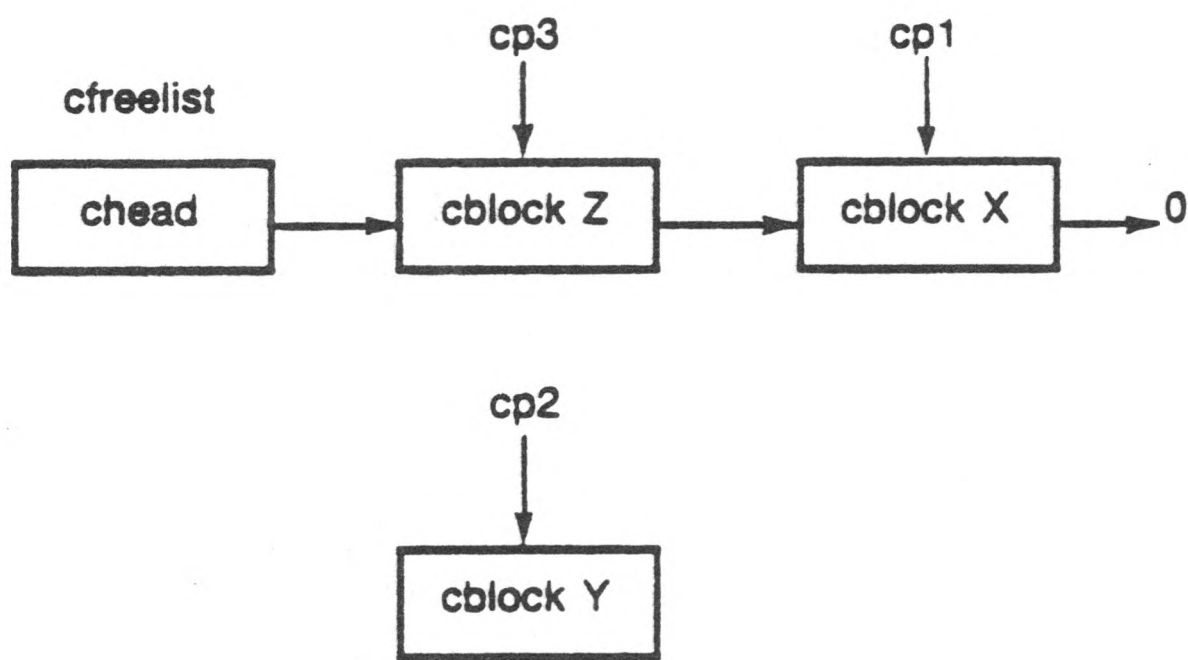
Initial state of *cfreelist* and variables `cp1`, `cp2`, And `cp3`



Result of putcf(cp1) ;



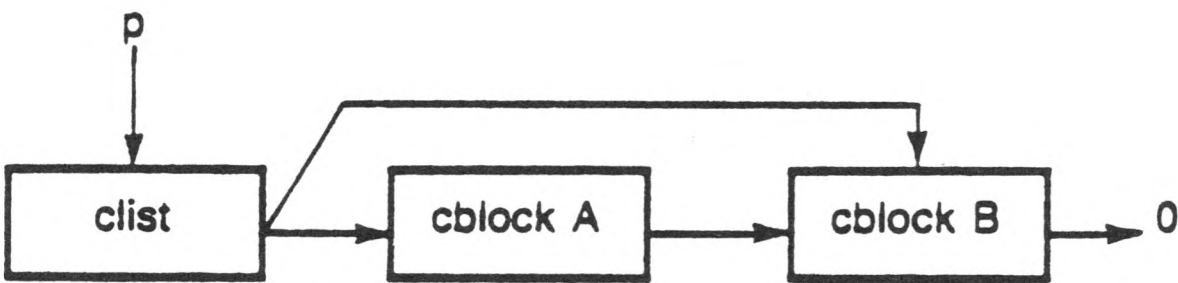
Result of putcf(cp3) ;



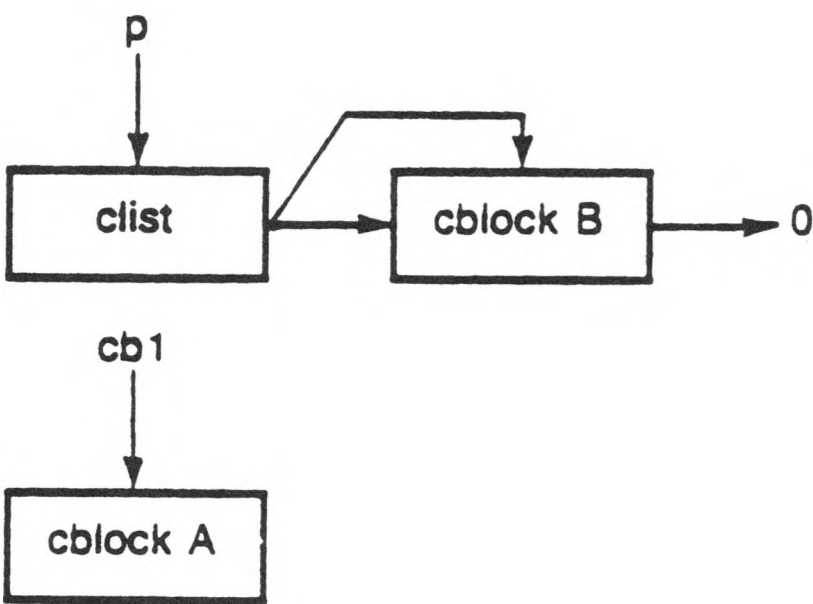
The putcf operation cannot return an error.

getcb

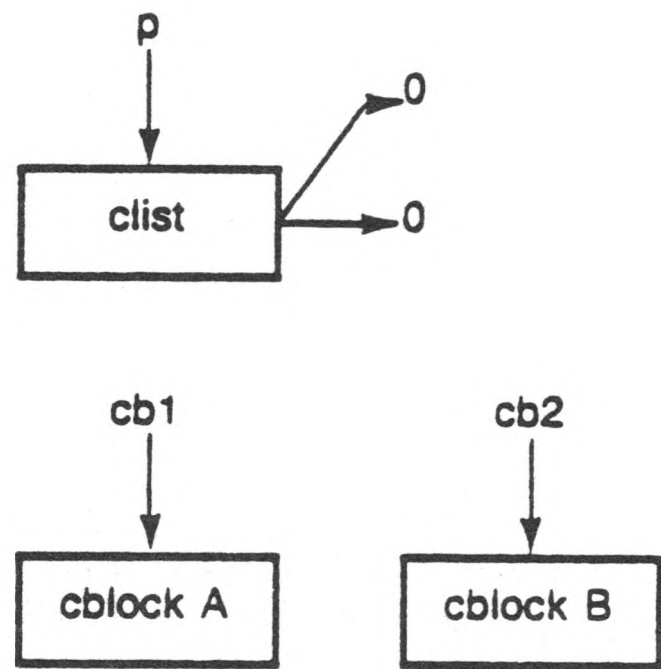
Initial state of clist p



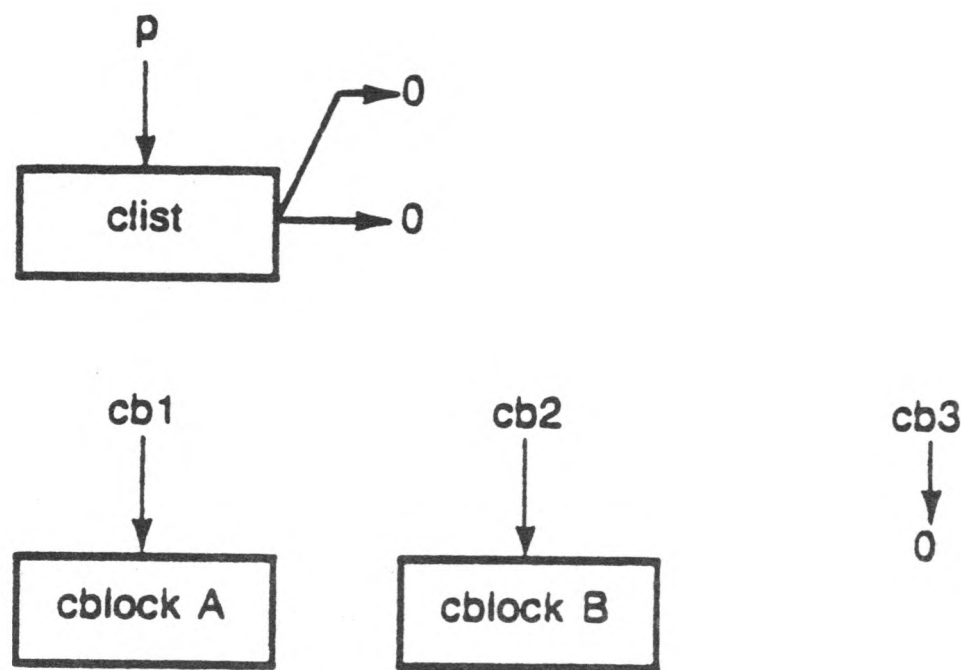
Result of `cb1 = getcb(p);`



Result of `cb2 = getcb(p) ;`



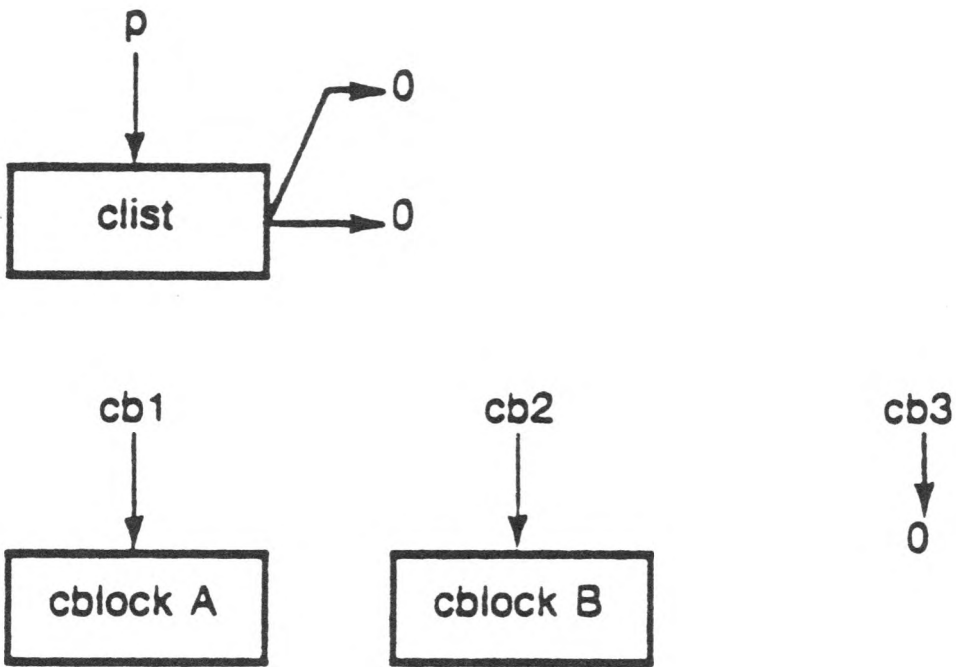
Result of `cb3 = getcb(p) ;`



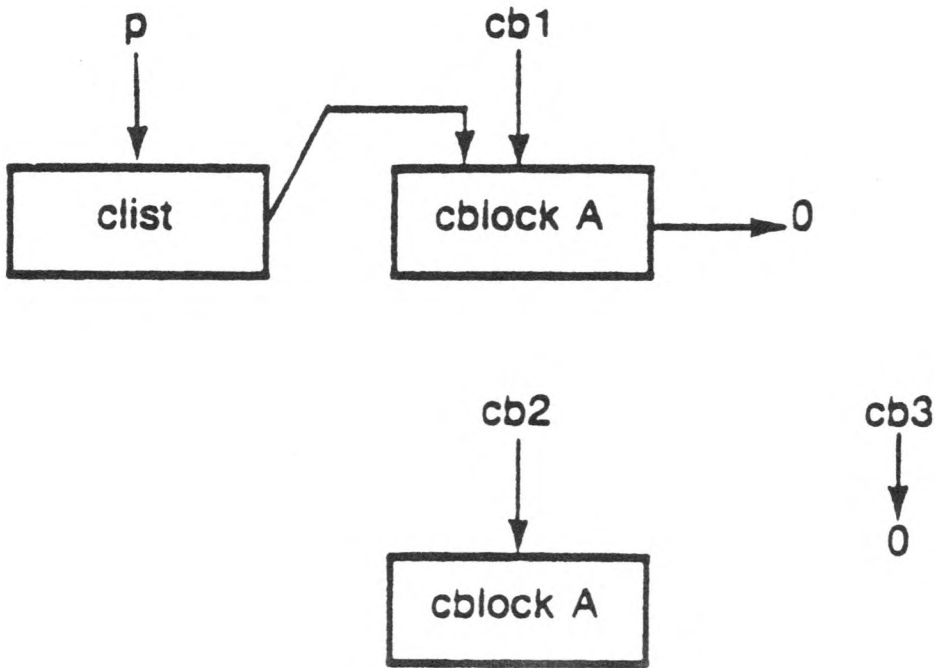
The value 0 is returned if there are no more *cblocks* on the *clist*.

putcb

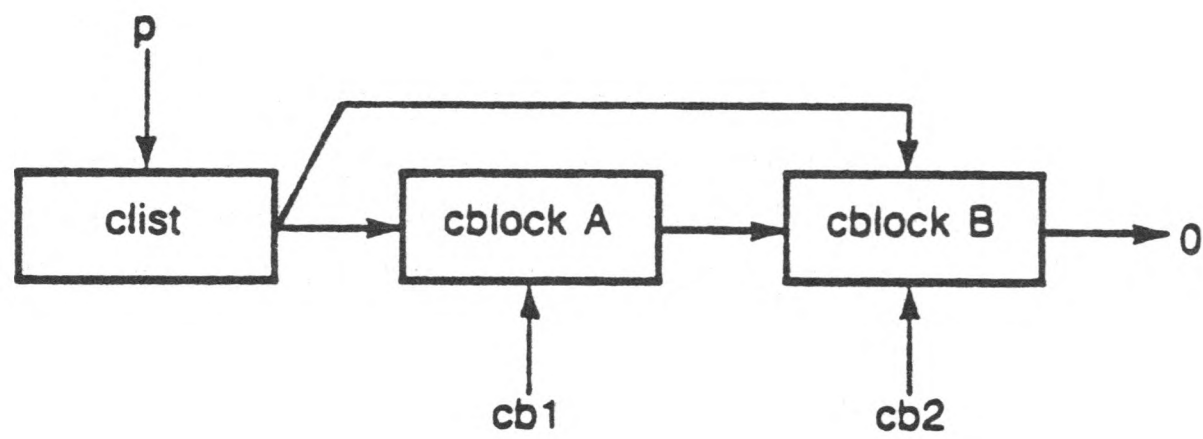
Initial state of clist p and variables cb1, cb2, and cb3



Result of putcb(cb1, p);



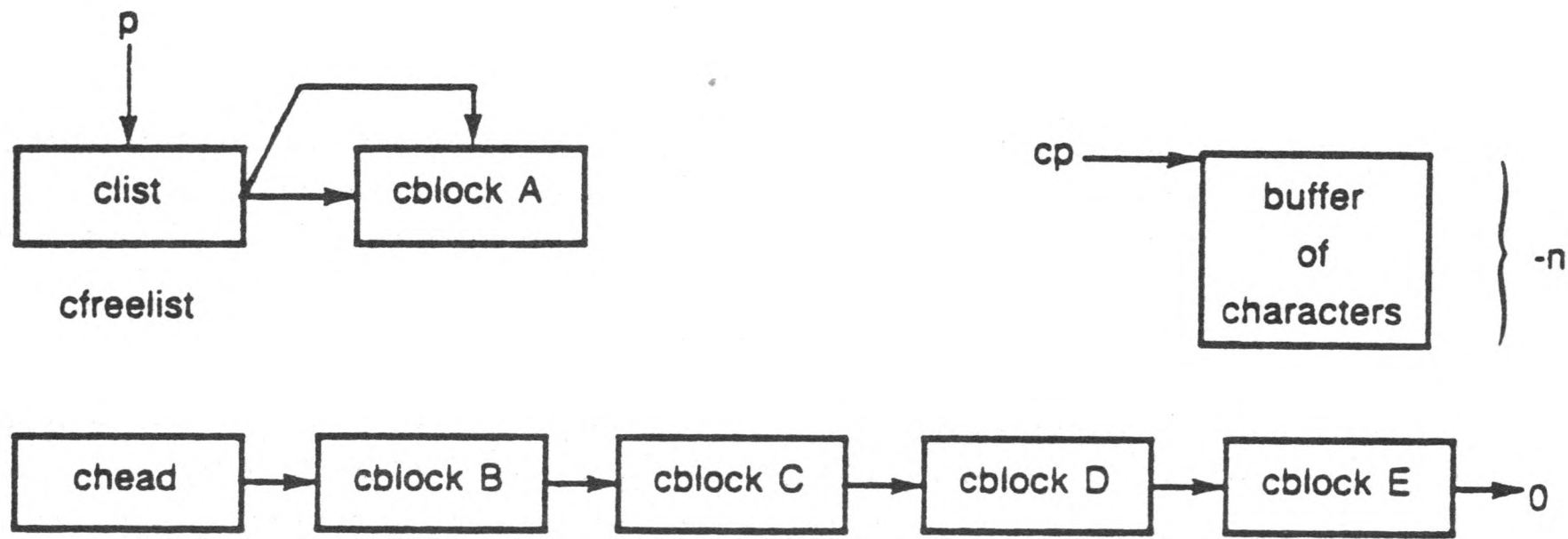
Result of `putc(cb2, p);`



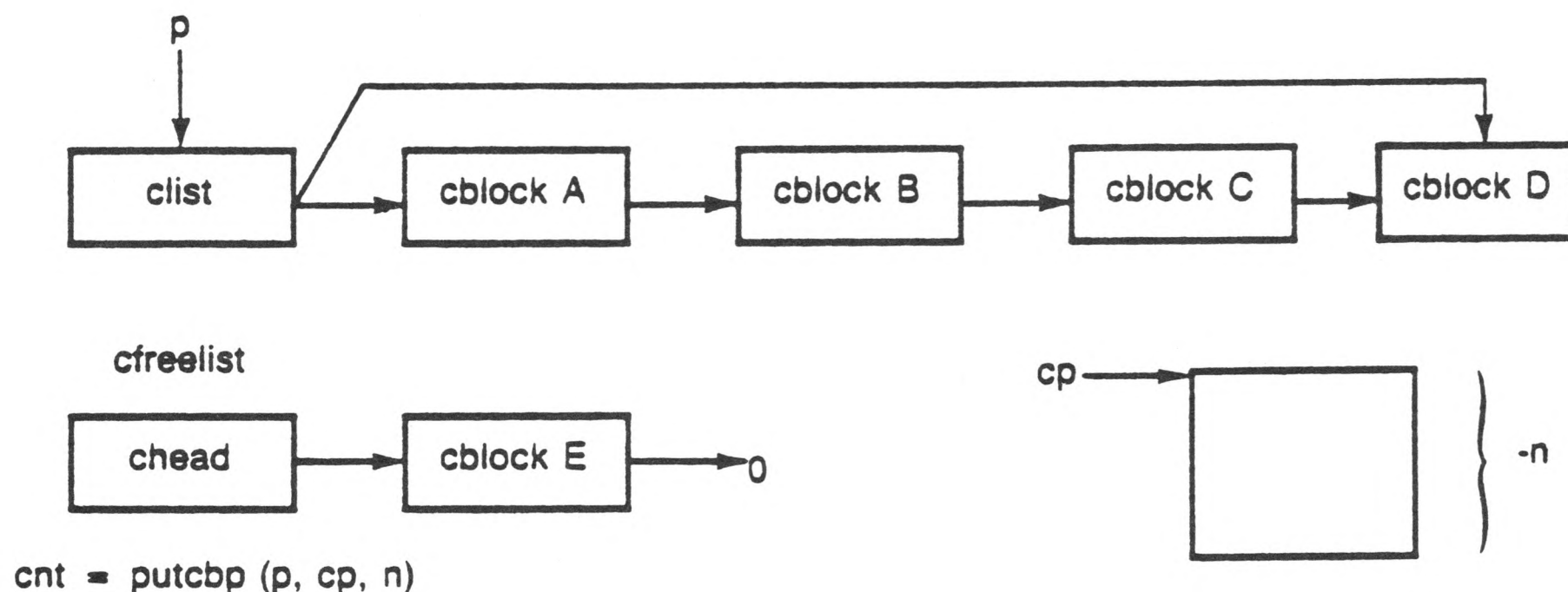
The *putc* operation cannot return an error.

`putcbp`

Initial state of `clist` `p`, `cfreelist`, and character buffer `cp`

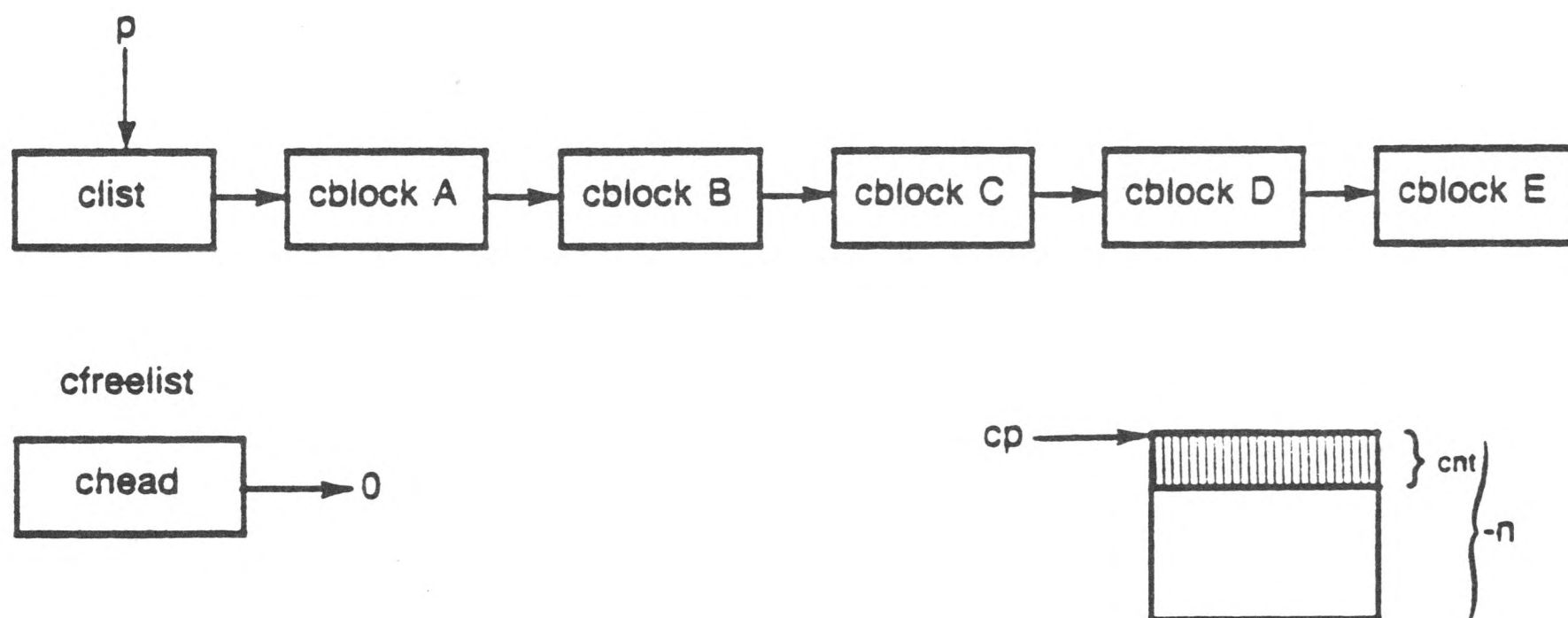


Result of $\text{cnt} = \text{putcbp}(p, cp, n)$



The following is attempted: Characters from *cp* are added to the *clist* (*p*). *Cnt* is the number of characters actually added to the *clist*. In this case, $\text{cnt} = n$.

Result of $\text{cnt} = \text{putcbp}(p, cp, n)$

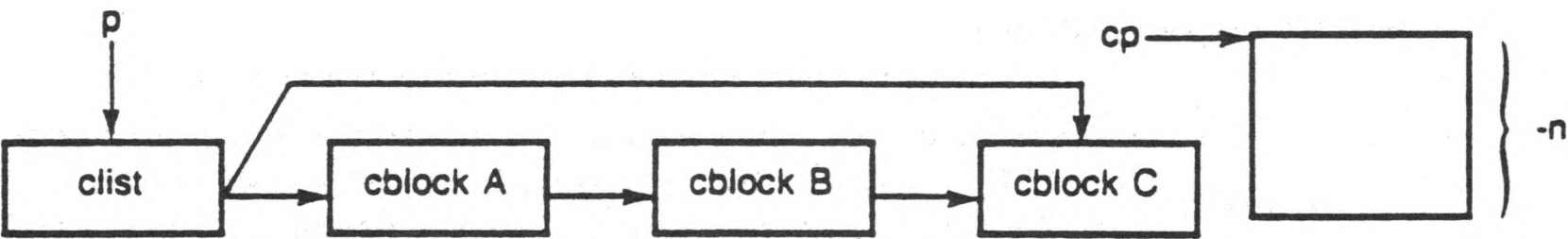


$\text{cnt} < n$ (Partial buffer transferred)

When *cfreelist* becomes empty, not all characters can be added. In this case *cnt* is less than *n*.

getcpb

Initial state of clist p and character buffer cp



Result of $\text{cnt} = \text{getc}(\text{p}, \text{cp}, \text{n})$



An empty buffer pointed to by *cp* is filled from the *clist*.

Result of $\text{cnt} = \text{getc}(\text{p}, \text{cp}, \text{n})$



Cnt records the number of characters actually moved to the buffer *cp*. When the *clist* becomes empty this will be less than *n*.

Appendix C.

Bootable Media

OVERVIEW

After a new kernel has been created, it can be loaded into the system (booted) and tested. Take these steps on the test system to boot a new kernel.

1. Perform a system shutdown. Logging in as "shutdown" performs an orderly shutdown.
2. Set the MANUAL/AUTO LOAD switch to the MANUAL position.
3. Reset the machine by engaging the RESET switch.
4. Choose the LOAD option from the Startup Menu.
5. Type in the media "device name" (h501, fd01, st01) and the full "pathname" of the kernel if required. This starts the load of the kernel. This step is slightly different depending on the media being used.
6. Enter <space bar> to begin execution of the kernel.

There are three different types of media that can be used to boot: hard disk, flex disk, and streaming tape. In the following examples, it's assumed that the kernel */driver/test.unix* needs to be tested.

HARD DISK BOOT

If the development system and the test system are the same, it is ideal to use the hard disk to load from. If the first 5.25 inch hard disk contains the file `/driver/test.unix`, the "device name" `h501` and the "pathname" `/driver/test.unix` are entered.

FLEX DISK BOOT

The flex disk can be used when the development system and the test system are different. The flex disk must be formatted with a bootblock and an Initial System Loader (ISL). This command formats the flex disk:

```
$ /etc/format -d /dev/rfdsk/0s0
```

A file system must be put on the disk using this command:

```
$ /etc/newfs -s 1 f501
```

Now the test kernel must be copied to the flex disk with these commands:

```
$ /etc/mount /dev/fdisk/0s0 /mnt/fd70
$ cp /driver/test.unix /mnt/fd70
$ /etc/umount /dev/fdisk/0s0
```

The flex disk can now be moved to the test system. With the LOAD switch on Manual, reset the system. When the device name is requested, enter `F501` or `F503` and then enter `/test.unix` for "pathname".

STREAMING TAPE

Streaming tape can also be used if the development system and the test system are different. This is faster than the flex method. The tape must have two files:

- A streaming tape boot block.
- The object of the kernel.

The object of the kernel must be extracted from the kernel *a.out(4)* file before copying to tape. The first 168 bytes of an *a.out* file are header information and must be stripped out. The following command strips out the unneeded information.

```
dd if=/driver/test.unix of=/driver/tape.unix ibs=168 skip=1
```

The streaming tape boot block and the stripped kernel can be put on the tape using the following two commands.

NOTE: The tape device must not rewind after copying the boot block or before copying the kernel object.

```
$ dd if=/sys/boot/boot.stp of=/dev/rstp/0yn
$ dd if=/driver/tape.unix of=/dev/rstp/0ny
```

The tape can be loaded on the first or second tape drive using the "device name" st01 or st02. No "pathname" is requested.

Appendix D.

Kernel Calls

OVERVIEW

This appendix is organized and formatted the same as the Programmer Reference. Each entry is alphabetic within its section.

NAME

intro - introduction to kernel calls

DESCRIPTION

This section describes the primary driver related kernel calls in alphabetical order. These calls execute in kernel user space and reside in the "C" libraries, lib0 -lib9.

intro
blt
copyin
copyout
cout
cpass
dmamalloc
dmapit
fubyte
fustr
fuword
getc
getcb
getcfc
geteslot
iodone
iomove
passc
physio
printf
putc
putcb
putcbp
putcfc
puterec
setjmp
sleep
spl
subyte
suword
timeout
wakeup

NAME

blt

SYNOPSIS

```
int blt(dest, src, count)
```

```
char *src;
```

```
char *dest;
```

```
int count;
```

DESCRIPTION

Move a block of data between src and dest with no checking for bus errors. This is an optimized memory copy routine.

RETURN VALUE

no meaning

COPYIN (k)

NAME

copyin

SYNOPSIS

```
int copyin(src, dest, count)
char  *src;
char  *dest;
int    count;
```

DESCRIPTION

Copy count bytes from src in user space to dest in kernel space.

RETURN VALUE

-1 bus error
otherwise success

NAME

copyout

SYNOPSIS

```
int copyout(src, dest, count)
char    *src;
char    *dest;
int     count;
```

DESCRIPTION

Copy count bytes from src in kernel space to dest in user space.

RETURN VALUE

-1 bus error
otherwise success

COUT (k)

NAME

cout

SYNOPSIS

```
coutb(register_addr, data)
char *register_addr;
char data
```

```
coutw(register_addr, data)
short *register_addr;
short data
```

```
coutl(register_addr, data)
int *register_addr;
int data
```

DESCRIPTION

These routines are used by some drivers to handle the writing to device registers. They simply copy the specified amount of data to the specified register except during power fail recovery when the routines are disabled and do nothing. Some devices behave disastrously if they have not been initialized before giving them commands. The critical out routines are used to protect against this.

RETURN VALUE

none

NAME

cpass

SYNOPSIS

int cpass()

DESCRIPTION

Return a character from user space. The source address is u.u_base. U.u_count is decremented. U.u_offset is incremented. U.u_base is incremented. Error checking is performed so user addresses do not have to be validated before using this function. If an addressing error occurs, u.u_error is set to EFAULT.

RETURN VALUE

0 The transfer is correct.

-1 Two possibilities

1. The value of u.u_count is 0 and no more characters can be transferred.
2. A fault has occurred and the value of u.u_error has been set to EFAULT.

DMAMALLOC (k)

NAME

dmamalloc

SYNOPSIS

```
int dmamalloc(bytes) int bytes;
```

DESCRIPTION

Allocate enough page blocks to map bytes number of bytes. The value returned from this function identifies the first page block and should be saved for use in the function dmapit. There are 64 pageblocks available for the 5000/20 and 256 for the 5000/40. If no page blocks are available, sleep until they become available.

RETURN VALUE

pageblock identifier

NAME

dmapit

SYNOPSIS

```
dmapit(segloc,pageloc,ladr,bytecount,ptepp)
int segloc      /* segment in context 31 allocated
                  at configuration */
int pageloc     /* page block reserved by
                  dmamalloc */
int ladr;       /* pointer to user space */
int bytecount   /* number of bytes to map */
int *ptepp;     /* pointer to user's page table
                  entry */
```

DESCRIPTION

The MMU described by segloc, pageloc, and bytecount is initialized with the physical addresses defined by ladr and ptepp. This call puts values in the segment registers for context 31 and the page registers in the identified pageloc.

RETURN VALUE

no meaning

FUBYTE (k)

NAME

fubyte

SYNOPSIS

```
int fubyte(src)
char  *src;
```

DESCRIPTION

Fetch user byte. Return one character from user space addressed by src.

RETURN VALUE

-1	bus error
1-255	success

NAME

fustr

SYNOPSIS

```
int fustr(src, dest, count)
char      *src;
char      *dest;
int       count;
```

DESCRIPTION

Copy a string between kernel and user space. Characters are copied from source to destination until a null character is transferred or until count number of characters have been transferred.

RETURN VALUE

```
-1 bus error
0  success
```


FUWORD (k)

NAME

fuword

SYNOPSIS

```
int  fuword(src)
int  *src;
```

DESCRIPTION

Fetch user word. Return one word from user space addressed by src.

RETURN VALUE

-1 bus error

otherwise success unless value returned from user space is -1

NAME

getc

SYNOPSIS

```
int getc(p) struct clist *p;
```

DESCRIPTION

Return a character from the clist pointed to by p. P is normally allocated by the user. This routine does suspend and may therefore be called from an interrupt service routine.

RETURN VALUE

-1 The clist p is empty.

0..255 value of the character which has been removed from the clist

GETCB (k)

NAME

getcb

SYNOPSIS

```
int getcb(p) struct clist * p;
```

DESCRIPTION

Take a cblock off the clist p. The address of the cblock is the return value of the function. The cblock returned can be used for any reason. When the cblock is no longer needed, it should be returned to the cfreelist using the function putcf. This routine does not sleep so it may be called by ISRs.

RETURN VALUE

0 clist p is empty

ptr any non zero value is a pointer to the cblock removed from p.

NAME

getcpb

SYNOPSIS

```
int getcbp(p, cp, n)
struct clist *p;
char *cp;
int n;
```

DESCRIPTION

Take a string of n characters off the clist p. The characters that are taken off the clist are moved to the location cp. The number of characters actually moved is returned. This routine does not sleep so it may be called by ISRs.

RETURN VALUE

n Number of characters actually removed from the clist (ie the number of characters in cp).

GETCF (k)

NAME

getcf

SYNOPSIS

struct cblock * getcf()

DESCRIPTION

Take a cblock off cfreelist. The address of the cblock is the return value of the function. The cblock returned can be used for any reason. When the cblock is no longer needed, it should be returned to the cfreelist using the function putcf. This routine does not sleep so it may be called by ISRs. If this routine fails (indicated by 0 return value), the caller must normally set cfreelist.c_flag = 1 and sleep(&cfreelist, priority).

RETURN VALUE

0 cfreelist is empty

ptr any non zero value is a pointer to the cblock

NAME

geteslot

SYNOPSIS

```
struct errhdr *geteslot(size)
int size;
```

DESCRIPTION

Return a pointer to a device error record of size bytes. The system error header is transparently added later. The return value is then type cast to the required error structure and its values filled. The function puterec is used to put the time and error type into an error header preceding the users error record.

RETURN VALUE

NULL No room left for error records.

otherwise The value returned is a pointer to a device error record.

IODONE (k)

NAME

iodone

SYNOPSIS

iodone (bp)
struct buf *bp;

DESCRIPTION

This is the only communication between the driver and the upper layer software which is initiated by the driver. When the drivers ISR indicates that the I/O is complete, this routine is called to indicate to the upper layers that it is finished.

RETURN VALUE

no meaning

NAME

iomove

SYNOPSIS

```
int iomove(kernel_buf, count, direction)
char      *kernel_buf;
int        count;
int        direction;
```

DESCRIPTION

Move a block of data between kernel and user space. The number of bytes specified by count is moved from kernel_buf to u.u_base if direction is B_WRITE. The movement is from u.u_base to kernel_buf if the direction is B_READ. B_WRITE and B_READ are defined in /usr/include/sys/buf.h. If u.u_segflg is 0, transfer is done with bus protection. If u.u_segflg is 1, transfer is done without bus protection. The following side effects of iomove are important:

u.u_base incremented by count

u.u_offset incremented by count

RETURN VALUE

-1 bus error

0 success

NAME

passc

SYNOPSIS

```
int passc(c)
char c;
```

DESCRIPTION

Pass a character into user space. The destination address is `u.u_base`. `U.u_count` is decremented. `U.u_offset` is incremented. `U.u_base` is incremented. Error checking is performed so user addresses do not have to be validated before using this function. If an addressing error occurs `u.u_error` is set to `EFAULT`.

RETURN VALUE

0 The transfer is correct.

-1 Two possibilities

1. The value of `u.u_count` just became 0 and no more characters can be transferred. -1 is returned when the last character was passed correctly not on a transfer that did not occur.
2. A fault has occurred and the value of `u.u_error` has been set to `EFAULT`.

physio

SYNOPSIS

```
int physio(strategy, bp, dev, flag)
int (strategy*)(); /* strategy function in the
                    driver */
struct buf * bp; /* buffer header */
dev_t dev; /* major and minor number
            */
int flag; /* B_READ or B_WRITE */
```

DESCRIPTION

Fill in the values in the buffer header using the user structure and then call `(strategy*)(bp)`. The values filled by `physio` are as follows:

b_dev	dev
b_flags	B_READ B_PHYS
b_count	u.u_count
un.baddr	u_base
b_proc	u.u_proc
b_blkno	calculated from u.u_offset

RETURN VALUE
no meaning

PRINTF (k)

NAME

printf

SYNOPSIS

The kernel version of printf is useful for debugging device drivers.

- Scaled down version of printf(3) described in the *Programmer Reference*.
- Uses polled output on the console terminal.
- The following conversion specifications can be used.

 %s string pointer

 %u unsigned integer

 %d decimal integer

 %o octal integer

 %x hexadecimal integer

 %D long decimal

- No field width specifiers allowed (ie %2s).

NAME

putc

SYNOPSIS

```
int putc(c, p)
char c;
struct clist *p;
```

DESCRIPTION

Put a character on the clist pointed to by p. P is normally allocated by the user. This routine does not suspend and may therefore be called from an interrupt service routine.

RETURN VALUE

- 1 Another cblock is required but cfreelist is empty.
- 0 The character has been put on the clist

PUTCB (k)

NAME

putcb

SYNOPSIS

int putcb(bp, p)

DESCRIPTION

Put a cblock on the clist p. The variable bp is a pointer to the cblock that needs to be placed on the list.

RETURN VALUE

none

NAME

putcbp

SYNOPSIS

```
int putcbp(p, cp, n)
struct clist *p;
char *cp;
int n;
```

DESCRIPTION

Put a string of n characters in the clist p. The characters that are put on the clist are moved from the location cp. The number of characters actually moved is returned. Fewer characters may be moved then requested if cfreelist becomes empty and the clist p becomes full. This routine does not sleep so it may be called by ISRs.

RETURN VALUE

n Number of characters actually removed from the clist (i.e., the number of characters in cp).

PUTCF (k)

NAME

putcf

SYNOPSIS

```
int putcf(bp)
struct cblock *bp;
```

DESCRIPTION

Put a cblock on cfreelist. The variable bp is a pointer to the cblock that needs to be returned. The cfreelist is a pool of cblocks shared by most character devices so unused blocks should be returned to cfreelist. If the value of cfreelist.c_flag is not zero, it is assumed that a process is sleeping, waiting for a cblock to be put on the list, and cfreelist.c_flag = 0 and wakeup(&cfreelist) is executed. This routine does not sleep so it may be called by ISRs.

RETURN VALUE

none

NAME

puterec

SYNOPSIS

```
puterec(err_rec, typ)
struct xyzerr *err_rec;
int type;
```

DESCRIPTION

Fill in the time stamp and type value in the system error header. Log the error record. Wakeup any processes waiting for error records.

RETURN VALUE

none

SETJMP (k)

NAME

setjmp

SYNOPSIS

setjmp(jb)

jmp_buf jb; /* jump buffer - array of 13
integers */

DESCRIPTION

A setjmp call causes jb to be initialized with the state describing a return from setjmp with a non-zero value.

- Nofault can be assigned to point to the first integer in jb.
- Thus when the bus error occurs execution resumes after the setjmp call.
- The first return from setjmp returns a 0 value.
- The second return from setjmp (the one caused by the bus error) returns a non-zero value.

RETURN VALUE

0 First immediate return from setjmp.
otherwise Second return from setjmp.

NAME

sleep

SYNOPSIS

```
int sleep(event, pri)
caddr_t  event;
int      pri;
```

DESCRIPTION

Move the currently executing process from the active (executing) state to the sleep state. Because ISRs are not associated with a predictable process, they should never call sleep. Event is a kernel address that uniquely identifies the reason for sleeping. The wakeup(event) kernel call wakes up all processes sleeping for the event. The processes issued the wakeup execute at the software priority pri. The most important effect of pri is that when $pri \leq PZERO$ a signal cannot disturb the sleep; if $pri > PZERO$ signals will be processed. Callers of this routine must be prepared for premature return and check that the reason for sleeping has gone away. If the priority (pri) has the value PCATCH or'ed into it, the sleep function returns the value of 1 if a software signal is sent to the process.

RETURN VALUE

- 0 Another process has issued a wakeup.
- 1 The sleeping process has been interrupted by a software signal.

NAME

spl

SYNOPSIS

```
void splx(newpri);
```

```
int spl0(), spl1(), spl2(), spl3();  
int spl4(), spl5(), spl6(), spl7();
```

DESCRIPTION

The routine `splx` changes the priority to any priority 0 through 7. `Spl0` changes the priority to 0. The rest of the `spln` where `n` is a number work similarly. This priority is the hardware processor priority. The hardware priority is also referred to as the interrupt level. When the processor is executing at interrupt level `n` only interrupts of value `n+1` or greater is recognized. Interrupts of value `n` or lower will be postponed until the processor priority is lowered. If the priority is changed, it is generally restored to its original value. Interrupting devices set the priority level of the cpu when generating an interrupt so the ISR runs at the set priority. The priority is restored when the ISR returns.

RETURN VALUE

- `splx(newpri)` returns no meaningful value.
- `spln()` where `n` is a number return the previous priority.

NAME

subyte

SYNOPSIS

int subyte(src, value)

char *src;

char value;

DESCRIPTION

Set user byte. Set the contents of src to value.

RETURN VALUE

-1 bus error

0 success

SUWORD (k)

NAME

suword

SYNOPSIS

int suword(src, value)

int *src;

int value;

DESCRIPTION

Set user word. Set the contents of src to value.

RETURN VALUE

-1 bus error

0 success

NAME

timeout

SYNOPSIS

```
timeout(isr_func, parameter, ticks);  
int (*isr_func)(); /* function to be called */  
int parameter;     /* parameter to be passed to  
                    isr_func */  
int ticks;         /* number of ticks before  
                    interrupt */
```

DESCRIPTION

The timeout function causes the following to be executed after number of system clock ticks. The actual function address and parameter value are stored in a delta list.

(*isr_func)(parameter)

RETURN VALUE

panic("Timeout table overflow");

occurs if the delta list overflows. No meaningful value is returned.

WAKEUP (k)

NAME

wakeup

SYNOPSIS

```
int wakeup(event)
caddr_t    event;
```

DESCRIPTION

Move all processes in the sleep state, sleeping on event, to the ready state.

Note: This one call may wakeup many processes.

RETURN VALUE

no meaning

Appendix E.

System Calls

OVERVIEW

This appendix is organized and formatted the same as the programmer reference. Each entry is alphabetic within its section.

Before using this appendix, read the introduction to the programmer reference for an explanation of the sections and the entries in each section.

The referenced operating system entries are in the operating system reference manuals.

CLOSE (2)

NAME

close - close a file descriptor

SYNOPSIS

```
int close (fildes)
int fildes;
```

DESCRIPTION

Close closes the file descriptor indicated by *fildes*, a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

FAILURE CONDITIONS

Close fails if *fildes* is not a valid open file descriptor. [EBADF]

RETURN VALUE

0 Successful completion
-1 Error; *errno* points to the error

SEE ALSO

creat(2), *dup*(2), *exec*(2), *fcntl*(2), *open*(2), *pipe*(2).

NAME

`ioctl` - control device

SYNOPSIS

```
ioctl (fildes, request, arg)
int fildes, request;
```

DESCRIPTION

Ioctl performs a variety of functions on character special files (devices). The pages describing the various devices in Section 7 of the Administrator Reference, UP-11761, discuss how *ioctl* applies to those devices.

FAILURE CONDITIONS

Ioctl fails if one or more of the following is true:

Fildes is not a valid open file descriptor.
[EBADF]

Fildes is not associated with a character special device. [ENOTTY]

Request or *arg* is not valid. See Section 7 of the Administrator Reference. [EINVAL]

RETURN VALUE

If an error has occurred, a value of -1 is returned and `errno` is set to show the error.

SEE ALSO

`termio(7)`.

LSEEK (2)

NAME

lseek - move read/write file pointer

SYNOPSIS

```
long lseek (fildes, offset, whence)
int fildes;
long offset;
int whence;
```

DESCRIPTION

Lseek sets the file pointer associated with *fildes* according to the value of *whence* as follows:

- 0 Set the pointer to *offset* bytes.
- 1 Set the pointer to its current location plus *offset*.
- 2 Set the pointer to the size of the file plus *offset*.

Upon successful completion, *lseek* returns the resulting pointer location as measured in bytes from the beginning of the file.

Fildes is a file descriptor returned from a *creat*, *open*, *dup*, or *fcntl* system call.

FAILURE CONDITIONS

Lseek fails and doesn't change the file pointer if one or more of the following is true:

Fildes is not an open file descriptor. [EBADF]

Fildes is associated with a pipe or fifo. [ESPIPE]

Whence is not 0, 1 or 2. [EINVAL and SIGSYS signal]

The resulting file pointer would be negative. [EINVAL]

WARNING

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

RETURN VALUE

Non-negative

integer Successful completion; the non-negative integer shows the file pointer value.

-1 Error; `errno` shows the error.

SEE ALSO

`creat(2)`, `dup(2)`, `fcntl(2)`, `open(2)`.

OPEN (2)

NAME

open - open for reading or writing

SYNOPSIS

```
#include <fcntl.h>
int open (path, oflag [ , mode ] )
char *path;
int oflag, mode;
```

DESCRIPTION

Open opens a file descriptor for the named file and sets the file status flags according to the value of *oflag* .

Upon successful completion a non-negative integer, the file descriptor, is returned.

Open sets the file pointer used to mark the current position within the file to the beginning of the file.

Open sets the new file descriptor to remain open across *exec* system calls. See *fcntl* (2).

Note that no process may have more than 64 file descriptors open simultaneously.

ARGUMENTS

Path points to a path name naming a file.

Oflag values are constructed by *or*-ing flags from the following lists:

Only one of these three flags can be used:

O_RDONLY

Open for reading only.

O_WRONLY

Open for writing only.

O_RDWR

Open for reading and writing.

Any number of the following flags may be *or*-ed into the *oflag*.

O_NDELAY

This flag may affect subsequent reads and writes. See *read* (2) and *write* (2).

When opening a FIFO:

If O_NDELAY is set and O_RDONLY is set, an *open* returns without delay.

If O_NDELAY is set and O_WRONLY is set, an *open* returns an error if no process currently has the file open for reading.

If O_NDELAY is clear and O_RDONLY is set, an *open* blocks until a process opens the file for writing.

If O_NDELAY is clear and O_WRONLY is set, an *open* blocks until a process opens the file for reading.

When opening a file associated with a communication line:

If O_NDELAY is set the *open* returns without waiting for carrier.

If O_NDELAY is clear the *open* blocks until carrier is present.

O_APPEND

If set, *open* sets the file pointer to the end of the file before each write.

O_CREAT

If the file exists, this flag has no effect. Otherwise, *open* creates the file and:

Sets the owner ID of the file to the effective user ID of the process

Sets the group ID of the file to the effective group ID of the process

Sets the low-order 12 bits of the file mode are set to the value of *mode* with the following modifications (see *creat* (2)):

clears all bits set in the file mode creation mask. See *umask* (2).

clears the *save text image after execution bit* of the mode. See *chmod* (2).

O_TRUNC

If the file exists, truncate its length to 0, and don't change the mode and owner.

O_EXCL If O_EXCL and O_CREAT are set, *open* fails if the file exists.

FAILURE CONDITIONS

Open fails and doesn't open the named file if one or more of the following is true:

A component of the path prefix is not a directory. [ENOTDIR]

O_CREAT is not set and the named file does not exist. [ENOENT]

A component of the path prefix denies search permission. [EACCES]

Oflag permission is denied for the named file. [EACCES]

The named file is a directory and *oflag* is write or read/write. [EISDIR]

The named file resides on a read-only file system and *oflag* is write or read/write. [EROFS]

The maximum (64) number of file descriptors are currently open. [EMFILE]

The named file is a character special or block special file, and the device associated with this special file does not exist. [ENXIO]

The file is a pure procedure (shared text) file that is being executed and *oflag* is write or read/write. [ETXTBSY]

Path points outside the allocated address space of the process. [EFAULT]

O_CREAT and O_EXCL are set, and the named file exists. [EEXIST]

O_NDELAY is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading. [ENXIO]

A signal was caught during the *open* system call. [EINTR]

The system file table is full. [ENFILE]

RETURN VALUE

Non-negative

integer Successful completion; the non-negative integer is the file descriptor.

-1 Error; *errno* shows the error.

SEE ALSO

chmod(2), *close*(2), *creat*(2), *dup*(2), *fcntl*(2), *lseek*(2), *read*(2), *umask*(2), *write*(2).

NAME

read - read from file

SYNOPSIS

```
int read (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

DESCRIPTION

Read attempts to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*. *Fildes* is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

On devices capable of seeking, the *read* starts at a position in the file given by the file pointer associated with *fildes*. *Read* increments the file pointer by the number of bytes actually read before returning.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a file is undefined.

Upon successful completion, *read* returns the number of bytes actually read and placed in the buffer; this number may be less than *nbyte* if the file is associated with a communication line (see *ioctl* (2) and *termio* (7)), or if the number of bytes left in the file is less than *nbyte* bytes. *Read* returns a value of 0 when an end-of-file has been reached.

When attempting to read from an empty pipe (or FIFO):

If *O_NDELAY* is set, the read returns a 0.

If *O_NDELAY* is clear, the read blocks until data is written to the file or the file is no longer open for writing.

When attempting to read a file associated with a tty that currently has no data available:

If `O_NDELAY` is set, the read returns a 0.

If `O_NDELAY` is clear, the read blocks until data becomes available.

FAILURE CONDITIONS

Read fails if one or more of the following is true:

Fildes is not a valid file descriptor open for reading. [EBADF]

Buf points outside the allocated address space. [EFAULT]

A signal was caught during the *read* system call. [EINTR]

RETURN VALUE

Non-negative integer	Successful completion; the non-negative number shows the number of bytes actually read.
----------------------	---

-1	Error; <code>errno</code> shows the error
----	---

SEE ALSO

`creat(2)`, `dup(2)`, `fcntl(2)`, `ioctl(2)`, `open(2)`, `pipe(2)`, `termio(7)`.

WRITE (2)

NAME

write - write on a file

SYNOPSIS

```
int write (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

DESCRIPTION

Write attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the *fildes*. *Fildes* is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

On devices capable of seeking, the actual writing of data proceeds from the position in the file shown by the file pointer. Upon return, *write* increments the file pointer by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

If the *O_APPEND* flag of the file status flags is set, *write* sets the file pointer to the end of the file prior to each write.

If the number of bytes to be written exceeds a defined limit (e.g., the *ulimit* (see *ulimit* (2)) or a physical limit (e.g., the physical end of a medium), *write* writes as many bytes as possible without exceeding the limit.

For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes returns 20. The next write of a non-zero number of bytes returns a failure.

If the file being written is a pipe (or FIFO), no partial writes are permitted. Thus, the write fails if a write of *nbyte* bytes would exceed a limit.

If the file being written is a pipe (or FIFO) and the `O_NDELAY` flag of the file flag word is set, then the writes to a full pipe (or FIFO) return a count of 0. Otherwise (`O_NDELAY` clear), writes to a full pipe (or FIFO) will block until space becomes available.

FAILURE CONDITIONS

Write fails and doesn't change the file pointer if one or more of the following is true:

Fildes is not a valid file descriptor open for writing. [EBADF]

An attempt is made to write to a pipe that isn't open for reading by any process. [EPIPE and SIGPIPE signal]

An attempt is made to write a file that exceeds the file size limit of the process or the maximum file size. See *ulimit* (2). [EFBIG]

Buf points outside the allocated address space of the process. [EFAULT]

A signal was caught during the *write* system call. [EINTR]

RETURN VALUE

Non-negative	Successful completion; the non-negative
integer	integer shows the actual number of bytes written.

-1	Error; <i>errno</i> shows the error.
----	--------------------------------------

SEE ALSO

creat(2), *dup*(2), *lseek*(2), *open*(2), *pipe*(2), *ulimit*(2).